Développement d'algorithmes d'imagerie et de reconstruction sur architectures à unités de traitements parallèles pour des applications en contrôle non destructif

Antoine Pedron

UNIVERSITE PARIS-SUD

ÉCOLE DOCTORALE : Sciences et Technologies de l'Information des Télécommunications et des Systèmes

***DISCIPLINE** : PHYSIQUE*

THÈSE DE DOCTORAT

Soutenue le 28/05/2013

présentée par

Antoine PEDRON

Développement d'algorithmes d'imagerie et de reconstruction sur architectures à unités de traitements parallèles pour des applications en contrôle non destructif

Rapporteur	Dominique HOUZET	Professeur, DIS, Grenoble-INP
Rapporteur	Jean-Luc LAMOTTE	Professeur, LIP6, Université Paris 6
Examinateur	Didier CASSEREAU	Maître de Conférences, LIP, Université Paris 6
Examinateur	Daniel ETIEMBLE	Professeur, LRI, Université Paris-Sud
Directeur de thèse	Lionel LACASSAGNE	Maître de Conférences, LRI, Université Paris-Sud
Encadrant CEA	Stéphane LE BERRE	Chef du LDI, CEA LIST/DISC

Remerciements

Le chemin que j'ai suivi m'a amené devant cette tâche, loin d'être la plus dure à mes yeux, mais que je considère comme l'une des plus importantes. La raison ? Des nombreux doctorants ou docteurs avec qui j'ai eu l'occasion de discuter, rares sont ceux qui se sentaient prédestinés à une faire une thèse initialement. C'est bien souvent les rencontres et discussions qui les auront amenés à entreprendre une telle aventure !

Je vais remonter à ma deuxième année de Master. A la recherche d'un stage de fin d'études original et représentant un défi, c'est Pierre et Yassir qui m'ont accordé leur confiance pour défricher avec eux l'univers émergeant des GPU. Je les remercie tout particulièrement puisqu'ils sont mon point d'entrée dans le monde de la recherche. C'est lors de ce premier passage au CEA que l'idée de faire une thèse a émergé chez moi ; Yncia, que je remercie pour ses nombreux éclairages en astrophysique, n'y est pas pour rien.

C'est à la suite de ce stage que j'intègre le laboratoire de mon futur directeur de thèse, dans un "bureau de thésards" où je vais passer 6 mois, côtoyant stagiaires, doctorants et docteurs en route vers leur HDR. Je salue amicalement Claude, Joël qui aura tenté quelques minutes après notre 1ère rencontre de mettre un peu de DSL dans mes GPU, Adrien et Pierre qui dans un registre différent chacun m'auront marqué par leurs qualités de jeunes chercheurs et bien sûr Tarik, sans qui je ne serais probablement pas à vous raconter tout cela - merci Niko.

La vraie aventure, parait-il, commence ici. Alors que j'étais à me demander s'il fallait que j'aille chercher du travail, un appel de Marie-Odile m'informe que je commence ma thèse dans 5 jours ! C'est l'occasion de te remercier pour ta réactivité et tes conseils en paperasseries diverses et variées ! Bien évidemment, je tiens à remercier les membres de la direction du DISC de m'avoir accueilli au sein de leur département. J'adresse mes profonds remerciements à l'ensemble des membres du LDI pour leur disponibilité.

J'en arrive à la passerelle et plus généralement à l'équipe de doctorants du DISC. Trois ans de thèse m'auront permis d'en voir passer un certain nombre ! Je vous remercie tous pour nos échanges et pour votre soutien les jours de galère. Je remercie plus particulièrement le Capitaine Passerelle et notre barde-doctorant, ainsi que les autres membres de l'équipe : Roberto, Warida, Souad, Mickaël, Adrien, Audrey, Clémence, Blandine, Marouane, Audrey, François, sans oublier Jason, petit dernier arrivé ! J'ai aussi une pensée pour les anciens, qui nous ont en quelque sorte passé le relai : Benoit, Sophie, Bo, Laura, Tekoing, Chiara et Amira ! Je remercie aussi bien sûr Victor et Adrien qui se reconnaitrons ! Du côté de l'université, je tiens à saluer l'implication de Guillaume ainsi que Jason (il est partout !) qui aura apporté plus que des idées dans mes travaux..

Un grand merci à Gilles, que je n'aurai pas laissé, ne serait-ce qu'une demie-journée tran-

quille sans une question sur CIVA ou le Contrôle Non Destructif... et éventuellement la course à pied que j'espère il pourra enfin reprendre sereinement.

Je tiens à exprimer ma reconnaissance à l'égard de Stéphane Le Berre, ainsi que Lionel Lacassagne de m'avoir fait confiance et de m'avoir permis de compléter ma formation universitaire de la plus parfaite des manières. Leurs conseils auront été précieux et m'auront permis de progresser significativement durant ces trois années.

J'adresse mes sincères remerciements à Dominique Houzet et Jean-Luc Lamotte pour m'avoir fait l'honneur de rapporter ma thèse et du temps qu'il y auront accordé. Je remercie aussi Didier Cassereau et Daniel Etiemble d'avoir accepté de faire partie de mon jury et de l'intérêt porté à mes travaux .

Je salue les collègues qui m'auront soutenu dans la dernière ligne droite. J'en arrive à mes amis pour qui j'ai une pensée sincère : vous m'avez toujours soutenu dans les moments difficiles et je vous en suis vraiment reconnaissant. Pensée aussi pour ma mère et mon frère bien évidemment, qui de loin, ont toujours cru en moi. Merci aussi à ma belle famille qui aura été un soutien important ! Je remercie aussi Jany, relectrice principale du manuscrit sans qui la lecture du manuscrit aurait probablement été terrible pour les membres du jury.

Je vais terminer en remerciement la personne qui mérite plus que n'importe qui mes remerciements. Celle qui est à mes côtés depuis maintenant 7 ans, ma femme, Victoria. Je te remercie pour ta patience, ton soutien, tes encouragements et bien d'autres choses. Sans toi, cette thèse ne serait peut-être pas arrivée à son terme de cette manière.

J'ai forcément oublié un certain nombre de personnes, j'en suis confus. Sachez simplement que vous avez tous été acteurs, d'une manière ou d'une autre, de ce travail et je vous en suis reconnaissant.

INTRODUCTION GÉNÉRALE

Le travail présenté dans ce manuscrit s'inscrit dans le contexte du Contrôle Non Destructif (CND) qui réunit un ensemble de méthodes et techniques utilisées pour inspecter et examiner des structures ou matériaux, sans les dégrader, durant la fabrication, leur utilisation ou lors de maintenances. Les domaines industriels concernés sont nombreux : l'énergie, les transports, le BTP ou encore la sidérurgie. Parmi les différentes méthodes existantes, les méthodes électromagnétiques (Courants de Foucault), la radiographie X et les ultrasons sont les plus exploitées. De manière générale, le CND vise à rechercher des défauts, c'est à dire mettre en évidence une variation des propriétés physiques de la pièce pouvant être critique à son usage. Pour se faire, des techniques d'imagerie et reconstructions ont été développées pour positionner et dimensionner ces défauts.

Le matériel évoluant, les volumes de données acquis lors des contrôles sont de plus en plus importants. Pour obtenir des temps de traitement rapides, il est nécessaire d'utiliser au mieux les architectures matérielles existantes. L'évolution de ces dernières ayant basculé depuis 2006, avec un arrêt de la montée en fréquence et la multiplication des unités de calculs, les algorithmes et codes existants ne sont plus adaptés à ces nouvelles architectures et n'exploitent que partiellement la puissance de calcul effectivement disponible.

La thèse s'inscrit dans le cadre de l'accélération d'algorithmes de reconstruction ultrasons développés par le Département d'Imagerie et Simulation pour le Contrôle (DISC) du CEA-LIST. Plus particulièrement, la plateforme logicielle CIVA regroupe un ensemble d'outils de simulation et d'analyse pour le CND. Jusqu'à aujourd'hui, ces algorithmes n'ont pas été développés pour profiter du parallélisme qu'offrent aujourd'hui la plupart des architectures matérielles. Il est donc nécessaire d'évaluer leur capacité à s'adapter à ce tournant technologique, en procédant à une analyse algorithmique détaillée, en évaluant à la fois les différentes architectures matérielles, ainsi que les différents langages et outils de programmation disponibles sur le marché. L'environnement d'exécution de la plateforme CIVA étant basé sur des machines de type station de travail, les architectures ciblées sont donc principalement les processeurs généralistes multicoeurs (GPP : *General Purpose Processor*) et les accélérateurs calculs tels que les cartes graphiques GPU (*Graphics Processing Unit*). Historiquement, les GPP ont largement contribué au parallélisme, par l'assemblage d'unités identiques intégrées dans la même puce ou des réseaux de processeurs (clusters). Concernant les GPU, à partir de 2006, Nvidia a décidé d'orienter son architecture, initialement dédiée à l'affichage, vers le calcul haute performance, en proposant un environnement de développement permettant de les programmer. Cet environnement propose un modèle de programmation basé sur l'exécution de milliers de threads de calcul en parallèle, très différent de celui proposé par les GPP.

Alors que la littérature scientifique s'est considérablement enrichie ces dernières années, en adaptation d'algorithmes pour ces architectures spécifiques ou encore en études comparatives de performances sur ces architectures parallèles, le domaine du CND par ultrasons reste encore peu fourni. Dans le domaine des ultrasons, plusieurs équipes de chercheurs se sont déjà intéressés à l'adaptation de différents calculs de simulation et de reconstruction sur des architectures GPU, utilisant des modèles simplifiés ou se limitant à des cas d'étude très restreints. Dans ces études, on constate qu'une partie des méthodes et traitements de reconstruction par ultrasons s'adapte bien à une parallélisation massive. Il est important de noter que malgré des points communs évidents entre l'échographie médicale et le CND par ultrasons, les caractéristiques des milieux considérés diffèrent de telle manière que de nombreuses approximations possibles dans le cas des échographies du corps humain, ne le sont plus dans le cas des pièces industrielles.

La thèse présentée dans ce manuscrit propose l'analyse de deux algorithmes très utilisés et nécessitant des temps de calculs problématiques pour une utilisation industrielle. La démarche a consisté tout d'abord à effectuer l'analyse algorithmique, optimisant si possible l'algorithme initial et étudiant sa capacité à être parallélisé. Dans un second temps, différentes implémentations ont été réalisées ciblant les architectures GPP et GPU, avec l'objectif de comparer au travers de benchmarks, à la fois les performances de ces architectures mais aussi des langages et outils de programmation utilisés. Les deux études réalisées ont permis de mettre en lumière un certain nombre d'éléments permettant de s'orienter vers des solutions supportant une parallélisation massive. Enfin, les algorithmes ont été intégrés sous forme de prototypes dans la plateforme CIVA, ce qui a permis de lever un certain nombre de problématiques notamment liées aux problèmes d'industrialisation des codes. Nous essayons, par les choix effectués, d'adresser certaines de ces problématiques, tout en prenant en compte le fait qu'elles se posent à une échelle plus large qu'est celle de l'intégration de code parallélisés et optimisés dans des plateformes logicielles industrielles.

Le manuscrit se décompose la manière suivante :

- Le chapitre 1 présente le contexte de la thèse. Celui-ci est constitué tout d'abord du domaine métier, partant des ultrasons et descendant jusqu'aux techniques de reconstruction, informations généralistes mais nécessaire à la compréhension des algorithmes étudiés. Ensuite, une synthèse de l'état de l'art des architectures parallèles et outils de programmation est effectuée afin de présenter les possibilités qui s'offrent à nous en vue de la parallélisation, et ce afin d'obtenir les meilleures performances possibles. Nous terminerons ce chapitre par la présentation de la méthode de mesure de performances utilisées. Quel que soit le type de reconstruction, il est nécessaire d'utiliser des données issues de la simulation, aussi simples soient-elles.
- Le chapitre 2 présente l'étude de l'algorithme Image Vraie Cumulée (IVC). Cet algorithme est générique et permet de repositionner les données ultrasonores dans le repère cartésien de la pièce. Cet algorithme est analysé pour une parallélisation massive sur différentes architectures matérielles et plusieurs implémentations ont été évaluées à l'aide d'un benchmark.
- Le chapitre 3 présente l'étude de l'algorithme de reconstruction de Focalisation en Tout Point (FTP). Ce type de reconstruction est appliqué à un certain type de contrôle ultrasons multiéléments et intègre un calcul de simulation. De la même manière que pour le chapitre 2, seront présentés l'analyse, la parallélisation sur différentes architectures et un benchmark.

TABLE DES MATIÈRES

CHAPITRE 1

CONTEXTE

Ce chapitre se décompose en quatre parties qui vont présenter le contexte de la thèse. En première partie, nous présenterons les bases des ultrasons, le fonctionnement des traducteurs ultrasonores et de la focalisation de faisceau pour ensuite entrer dans le détail du CND par ultrasons et nous terminerons par une brève présentation du logiciel CIVA. Dans une seconde partie, nous présenterons les architectures parallèles pouvant être exploitées sur une machine de type station de travail. Dans une troisième partie, nous ferons un récapitulatif des différents outils de programmation à notre disposition pour programmer ces architectures. Nous terminerons ce chapitre par une discussion sur les différentes problématiques d'intégration que l'on peut rencontrer lorsque l'on intègre des codes optimisés et parallélisés dans un logiciel industriel tel que CIVA, ainsi que quelques remarques sur les mesures de performances.

1.1 Le contrôle non destructif par ultrasons

1.1.1 Les ultrasons

Les ultrasons sont des ondes mécaniques de fréquence supérieure à la limite de l'audition de l'oreille humaine donc environ supérieure à 15 kHz. En règle générale, la vitesse de propagation d'une onde est d'autant plus grande que le temps de transmission de l'information d'une particule élémentaire constituant la matière à sa voisine est rapide, suivant le niveau de cohésion de la matière. Ainsi, les vitesses de propagation d'une onde dans l'air, l'eau ou l'aluminium sont respectivement approximativement égales à 340 $\mathrm{m.s^{-1}}$, 1500 $\mathrm{m.s^{-1}}$et 6000 $\mathrm{m.s^{-1}}$. Selon la nature du milieu (fluide ou solide), une onde est entièrement décrite par une grandeur scalaire comme la pression et non par une grandeur vectorielle comme le déplacement particulaire. Dans le cas d'un milieu fluide, on parlera d'ondes acoustiques, alors que l'on désignera des ondes élastiques pour un milieu solide. Hormis le CND que nous détaillerons par la suite, il existe beaucoup d'applications des ultrasons notamment dans le domaine médical ou en acoustique sous-marine (sonar).

1.1.1.1 Types de propagation d'une onde ultrasonore

Dans un milieu fluide, seules peuvent se propager des ondes de compression-dilatation : l'onde se propage en comprimant puis en détendant la matière de proche en proche (figure 1.1 à droite). Ces ondes sont nommées ondes longitudinales. Dans un milieu isotrope (mêmes caractéristiques dans toutes les directions), deux types d'ondes peuvent se propager : des ondes de compression et des ondes de cisaillement. Les premières sont identiques à celles décrites précédemment en milieu fluide. Les secondes, liées à des contraintes de cisaillement, se propagent sans variation de volume. Ces ondes sont nommées ondes transversales (figure 1.1 à gauche). Dans les métaux par exemple, leur vitesse de propagation est environ deux fois plus faible que la vitesse de propagation des ondes longitudinales.

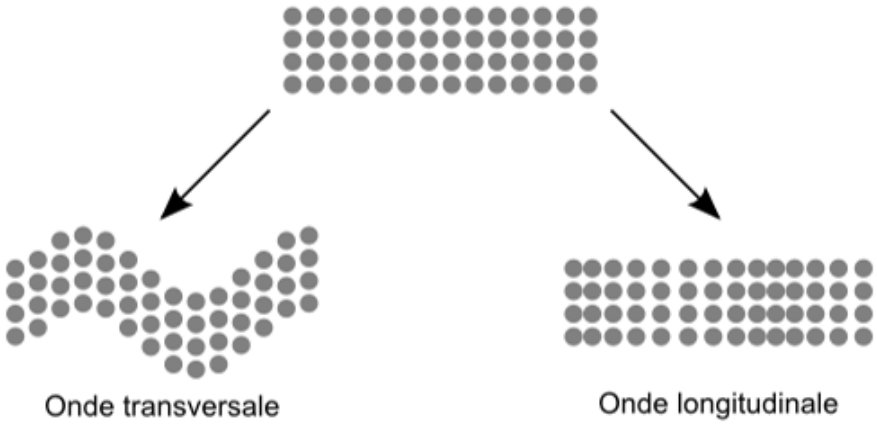


FIGURE 1.1 – Types de propagation des ondes ultrasonores.

1.1.1.2 Interaction d'une onde avec une interface plane

Dans le cas où deux milieux sont séparés par une interface plane, la nature des ondes réfléchies et transmises à l'interface dépend de la nature de l'onde incidente (longitudinale ou transversale) et de la nature des milieux (fluides ou solide).

En CND, le cas le plus courant correspond à un traducteur fonctionnant en immersion oblique ou au contact rayonnant des ondes longitudinales dans le sabot et ce, au travers d'une interface fluide/solide tel que sur la figure 1.2. Dans ce cas, une onde longitudinale incidente rencontrant l'interface selon un angle θ_i, donne potentiellement naissance à 3 ondes : une onde longitudinale réfléchie avec le même angle d'incidence par rapport à la normale de l'interface et deux ondes réfractées, une onde longitudinale se propageant avec un angle θ_{tL} et une onde transversale d'angle θ_{tT}. La relation de Snell-Descartes [Royer. and E.Dieulesaint, 1996], basée sur la conservation de la phase, lie les directions de ces différentes ondes.

$$\frac{\sin\theta_i}{c_{eau}} = \frac{\sin\theta_r}{c_{eau}} = \frac{\sin\theta_{tL}}{c_{Lactier}} = \frac{\sin\theta_{tT}}{c_{Tacier}} \tag{1.1}$$

Cette dernière relation montre que l'angle de réfraction de l'onde transmise augmente avec la vitesse des ondes dans le milieu de propagation. Ainsi, l'angle de propagation d'une onde

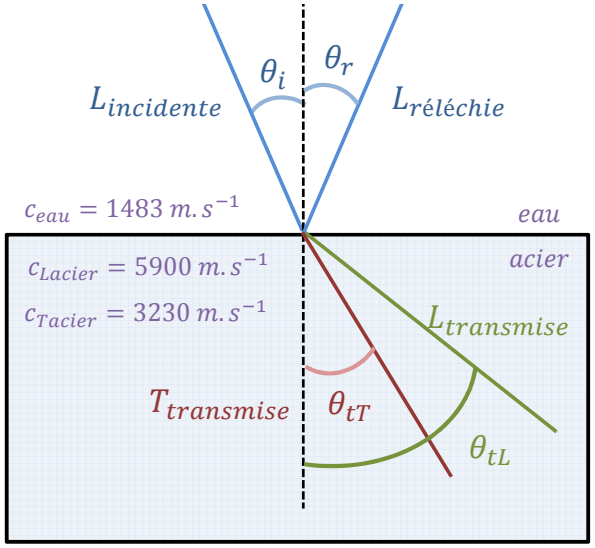


FIGURE 1.2 – Cas d'une interface eau/acier pour une onde longitudinale incidente (configuration courante en CND).

transversale sera plus faible que celui d'une onde longitudinale, de vitesse supérieure. Selon l'angle d'incidence dans le milieu fluide, nous aurons plusieurs possibilités : soit aucune onde propagée dans le second milieu, soit l'onde transversale seule, soit deux ondes. Si l'angle d'incidence dépasse le premier angle critique défini par :

$$\theta_{Lcrit} = \arcsin \frac{c_{Lacier}}{c_{eau}} \tag{1.2}$$

seule l'onde transversale est propagée dans le second milieu.

1.1.2 Les traducteurs ultrasonores

Un traducteur est défini comme le dispositif électro-acoustique qui incorpore généralement un ou plusieurs transducteurs destinés à l'émission et/ou à la réception des ondes ultrasonores. Ainsi, un transducteur est un élément actif du traducteur permettant de convertir l'énergie électrique en énergie acoustique et inversement.

1.1.2.1 Emission et réception des ondes ultrasonores

Il existe différents procédés permettant de générer et/ou de détecter des ondes ultrasonores, que ce soit au moyen de lasers, par effet électromagnétique (effet inductif) ou électrostatique (effet capacitif), ou encore par magnétostriction, mais le procédé le plus largement utilisé fait appel à l'effet piézoélectrique découvert par les frères Curie [Curie and Curie, 1880]. Le composant principal est un élément piézoélectrique (lame mince pour les transducteurs dans la gamme du mégahertz et au-dessus). A l'émission, l'élément piézoélectrique est mis en vibration par un signal électrique (effet piézoélectrique inverse). A la réception, la vibration ultrasonore créée un champ électrique (effet piézoélectrique direct) détecté sur des électrodes situées de part et d'autre de la lame. Ces transducteurs sont utilisables aussi bien en émission qu'en réception.

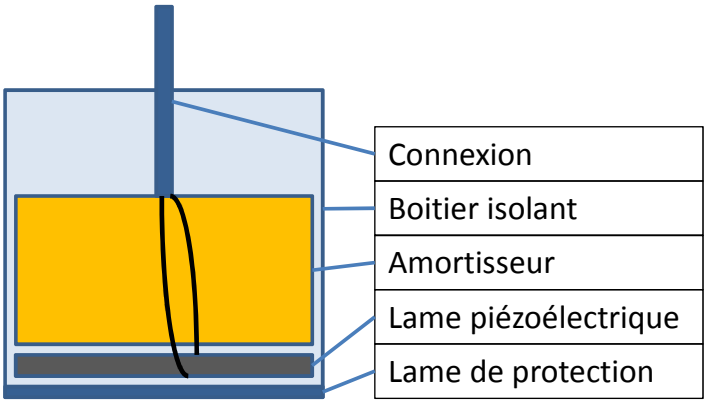


FIGURE 1.3 – Eléments constituants d'un traducteur ultrasonore.

La figure 1.3 représente un schéma d'un tel traducteur et de ses éléments principaux. Les ultrasons étant très fortement atténués dans l'air et, pour ce type de transducteurs étant générés à l'intérieur du traducteur, il est en général nécessaire d'utiliser un milieu de couplage entre le traducteur et l'élément que l'on veut sonder. Lors d'un contrôle industriel, l'inspection sera réalisée, soit par une immersion de la pièce dans un fluide (immersion parfois locale, généralement de l'eau), soit par contact en interposant un couplant (gel) entre la pièce et le traducteur.

1.1.2.2 Différents types de focalisation

La focalisation de faisceaux émis par les traducteurs permettent de concentrer l'énergie acoustique dans une zone, ce qui donne la possibilité d'accroitre la résolution et la sensibilité des signaux. Divers moyens peuvent être utilisés :

- **La lentille acoustique** : à l'instar de ce qui est réalisé en optique, une lentille peut être placée directement sur la face du transducteur. Néanmoins, contrairement à l'optique, la focalisation nécessite une lentille concave pour obtenir une convergence du faisceau, car le matériau utilisé pour la lentille a généralement une vitesse de propagation plus grande que le milieu de propagation dans lequel vont être créées les ondes. La focale obtenue est donc directement proportionnelle au rayon de courbure de la lentille.
- **Lame piézoélectrique mise en forme** : les avancées technologiques en matière de capteurs, piézocomposites en particulier, permettent la mise en forme directe du capteur, ce qui permet de s'affranchir des inconvénients des traducteurs à lentille (figure 1.4).
- **Focalisation électronique par loi de retard** : une autre technique qui permet d'imiter le fonctionnement d'une lentille de manière électronique sur un traducteur multiéléments. Pour un point de focalisation donné, les trajets entre chaque élément du traducteur et le point de focalisation sont déterminés, puis les différences de temps de trajets sont compensées de manière électronique en retardant l'émission des tirs de sorte que l'ensemble des éléments contribuent en phase et de manière collective au point de focalisation. Inversement, les signaux reçus peuvent être décalés temporellement avant sommation afin d'optimiser la détection (figure 1.5).

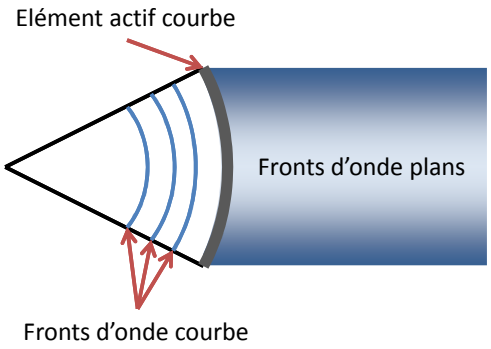


FIGURE 1.4 – Lame piézoélectrique mise en forme.

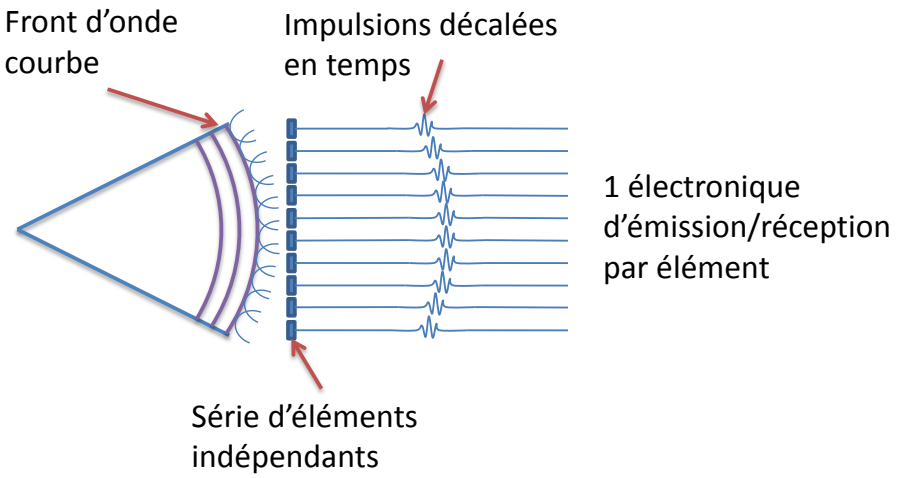


FIGURE 1.5 – Lame piézoélectrique mise en forme.

1.1.2.3 Différents types de traducteur

Selon le matériau, la géométrie de l'élément sondé, les défauts recherchés et la configuration expérimentale requise, différents types de traducteurs sont utilisés. La liste qui suit n'est donc pas exhaustive.

- **Les traducteurs en immersion** fonctionnent dans l'eau pour réaliser le couplage acoustique entre le traducteur et la pièce (figure 1.6).
- **Les traducteurs au contact** utilisent un sabot en plexiglas pour réaliser le couplage avec la pièce (figure 1.7).

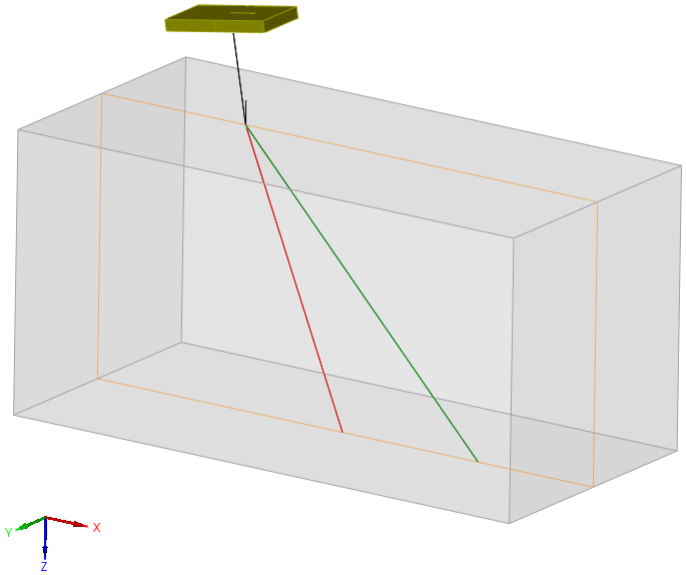


FIGURE 1.6 – Traducteur en immersion.

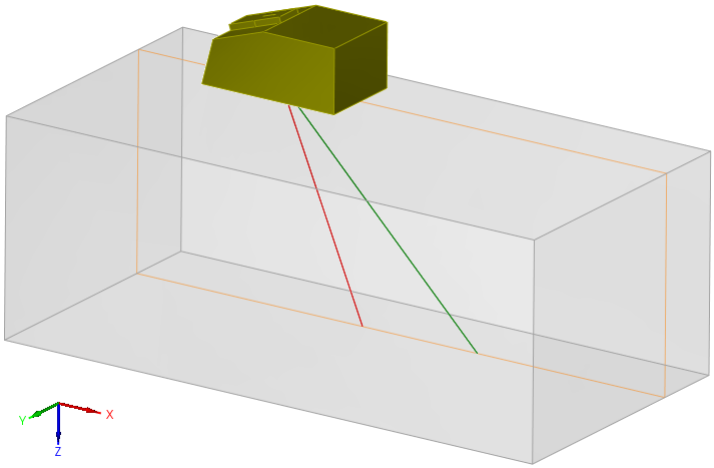


FIGURE 1.7 – Traducteur au contact.

- **Les traducteurs à émission/réception séparées** distinguent la fonction d'émission de la fonction de réception et comportent donc deux élément sensibles (figure 1.8). Ce type de dispositif permet de s'affranchir des échos parasites (échos de sabot) susceptibles d'apparaitre dans des capteurs fonctionnant au contact composés de mêmes éléments en émission et réception. Ce sont des échos parasites provenant des échos générés dans le sabot, réfléchis à l'intérieur du sabot et reçu par le capteur en émetteur-récepteur.
- **Les traducteurs multiéléments** sont des réseaux d'éléments indépendants. L'application de lois de retards sur l'ensemble ou une partie des éléments permet de maitriser le faisceau ultrasonore rayonné. Tous les types de traducteurs définis précédemment peuvent être conçus en multiéléments. La découpe du traducteur définit la maitrise du faisceau (figure 1.9). Une évolution de ces traducteurs est développée au CEA : il s'agit de traducteurs flexibles permettant de rester au contact de surfaces à géométries complexes. Il suit les déformations de l'interface et mesure en temps réel embarqué celles-ci, permettant

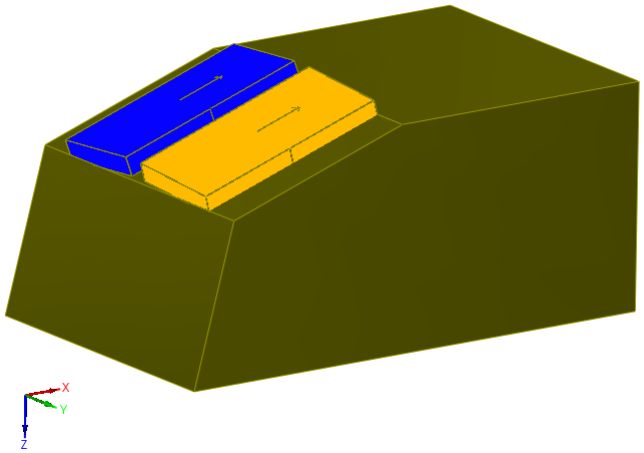


FIGURE 1.8 – Traducteur SE (fonctions d'émission et réception séparées).

l'application de lois de retard adaptées permettant de la compenser (figure 1.10).

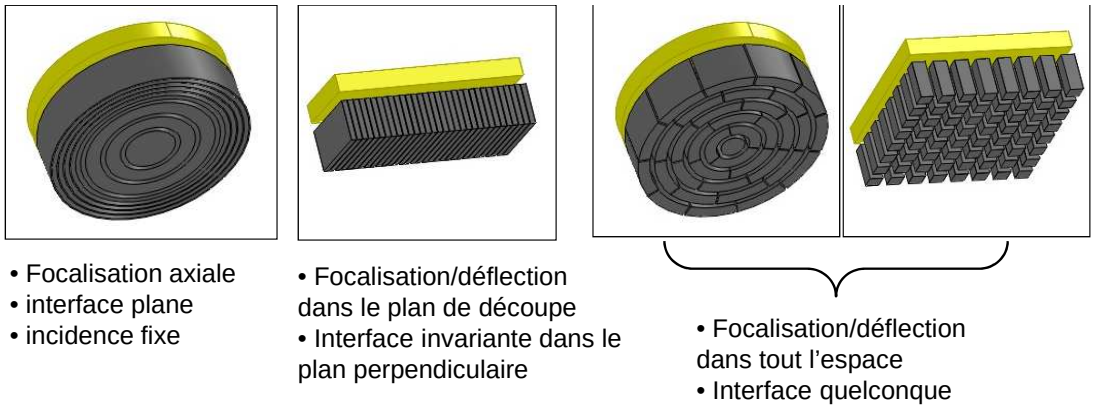


FIGURE 1.9 – Découpes de traducteurs multiéléments.

1.1.3 Le contrôle non destructif par ultrasons et méthodes de reconstruction

Le contrôle non destructif par ultrasons (CND US) est couramment utilisé pour rechercher des défauts dans des pièces en production, utilisation ou maintenance. Le principe de base est le suivant : un signal électrique est envoyé au transducteur ultrasonore, qui émet dans la structure à contrôler une onde élastique dont la propagation sera influencée par la présence d'éventuels défauts (fissures, inclusions,...). Après avoir traversé la partie à contrôler, l'onde est détectée par un récepteur ultrasonore. Cette onde est alors convertie en signal électrique. Cette mesure et son interprétation doivent permettre de déduire la présence ou non de défauts ainsi que leur nature et leur dimension. Ce principe simple met en jeu des phénomènes physiques telles que la diffraction de la source, la réflexion/réfraction aux différentes interfaces, l'interaction de l'onde avec le défaut, qui peuvent rendre l'interprétation des mesures difficiles.

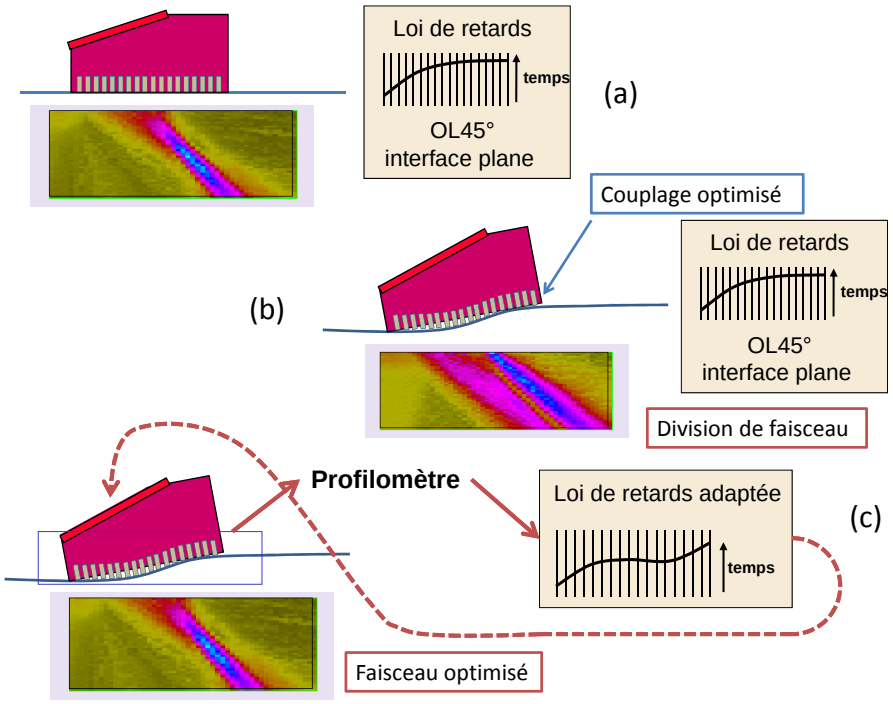


FIGURE 1.10 – Traducteur confortable : a) interface plane, b) interface complexe avec loi interface plane, c) interface complexe avec loi adaptée.

1.1.3.1 Exemple de contrôle en CND US

Soit une pièce d'épaisseur plongée dans un fluide, un traducteur émet une onde ultrasonore dans le fluide. Une partie de cette onde va pénétrer dans la pièce et l'autre partie va être réfléchie par la première interface (eau/acier), puis reçue par le transducteur. Ce dernier sert donc à la fois d'émetteur et de récepteur. C'est ce que l'on appelle un fonctionnement en échographie. Le premier écho reçu par le transducteur est appelé écho de face avant ou écho d'interface. Si l'onde ayant pénétré dans la pièce ne rencontre pas de défaut, elle va se propager jusqu'à la dernière interface. Une partie de l'onde est réfléchie par cette interface (acier/eau) pour revenir vers le transducteur. L'écho reçu correspond donc à un aller-retour de l'onde dans la pièce et prend le nom d'écho de fond.

Dans le cas où l'onde rencontrerait un défaut, ce dernier va servir de réflecteur et un écho supplémentaire appelé écho de défaut sera reçu par le transducteur. Connaissant la vitesse de propagation, une mesure de temps permet de connaitre la position du défaut dans la pièce (figure 1.11).

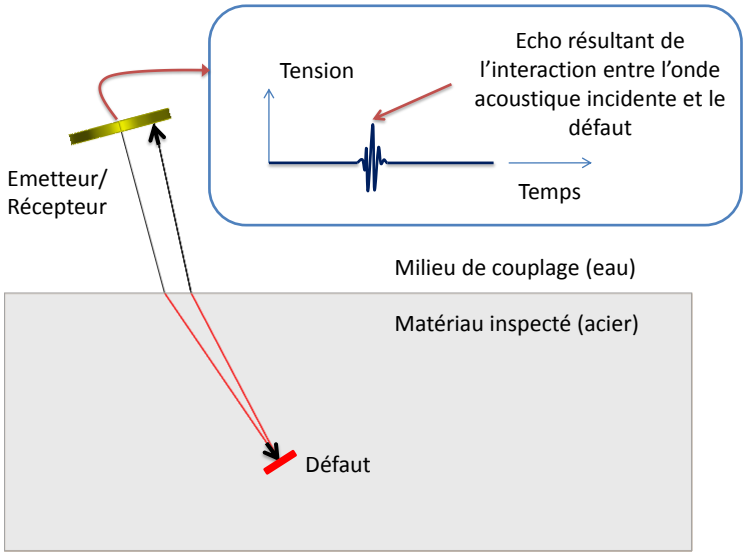


FIGURE 1.11 – Principe de l'échographie en CND par ultrasons.

1.1.3.2 Représentation des données d'un contrôle ultrasons

A partir de l'ensemble des signaux temporels acquis par le traducteur au cours du contrôle, nous pouvons extraire les différentes vues, A-scan, B-scan, C-scan et D-scan (figure 1.12 et 1.13).

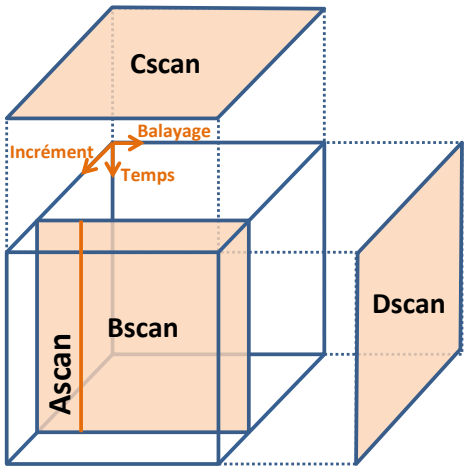


FIGURE 1.12 – Assemblage des différentes vues d'un volume de données d'un contrôle ultrasonore classique.

- **A-scan** : représente le signal reçu par le traducteur en une position d'acquisition. L'amplitude reçue est représentée en fonction du temps. Cette forme d'onde est l'élément de base des représentations des données ultrasonores.
- **B-scan** : résulte de la juxtaposition de l'ensemble des A-scans d'une ligne de balayage.

L'amplitude du signal est codée selon une échelle de couleurs ou de niveaux de gris. En abscisse est représenté le balayage, en ordonnées le temps de vol. Cette vue est assimilable à une vue de coupe de la pièce.

- **C-scan** : représente, sur une image 2D suivant les axes de balayages et d'incréments, la somme ou le maximum d'amplitude de chaque A-scan. Cette vue est assimilable à une vue de dessus de la pièce.
- **D-scan** : correspond au même volume de données que le C-scan. La sommation ou le maximum d'amplitude est pris sur l'axe des incréments et non des temps. Il en résulte une représentation 2D suivant les axes incréments et temps.

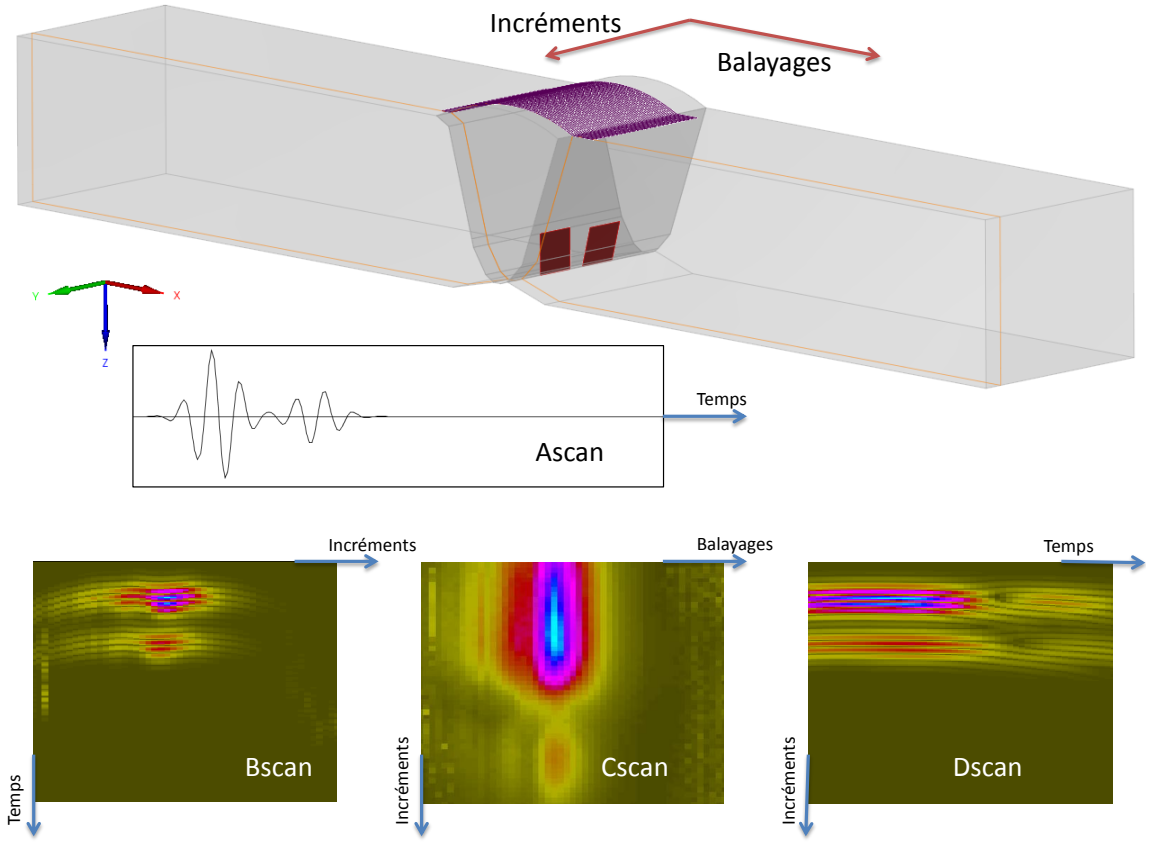


FIGURE 1.13 – Représentation des données ultrasonores échographiques.

1.1.3.3 Techniques d'inspection en CND par ultrasons

Il existe de nombreuses techniques d'inspection par ultrasons permettant d'utiliser au mieux les traducteurs pour couvrir un maximum du volume que l'opérateur souhaite contrôler. Nous allons brièvement présenter deux techniques communément utilisées en CND US.

- **La commutation électronique** : Les traducteurs multiéléments permettent d'ajouter, voire dans certains cas de substituer au balayage mécanique du capteur, un balayage électronique qui permet des cadences d'acquisition beaucoup plus élevées qu'un déplacement mécanique. Le principe est d'utiliser un groupe d'éléments actifs (par exemple 16 éléments d'un réseau en comportant 64) en émission-réception, puis de décaler les éléments actifs.

- **Le balayage angulaire** : lors d'un contrôle par ultrasons classique, afin de détecter des fissures se propageant généralement perpendiculairement à la pièce, le traducteur est incliné afin de générer un faisceau à un angle donné dans la pièce et déplacé à la surface de celle-ci. Pour les multiéléments, l'inclinaison peut être obtenue par loi de retard. Le contrôle par multiéléments autorise également à réaliser un contrôle à des angles différents (tirs) pour une même position. Des algorithmes de repositionnement sont fréquemment utilisés pour représenter les données expérimentales issues de ce type de contrôle. Ils vont être présentés par la suite.

1.1.3.4 Méthodes de reconstruction

L'objectif de la reconstruction est de localiser et de caractériser les défauts à l'origine des échos détectés. Les différentes méthodes ont pour point commun d'associer aux données expérimentales des données de simulées et de produire une image dans le repère cartésien de la pièce (c.f. figure 1.14).

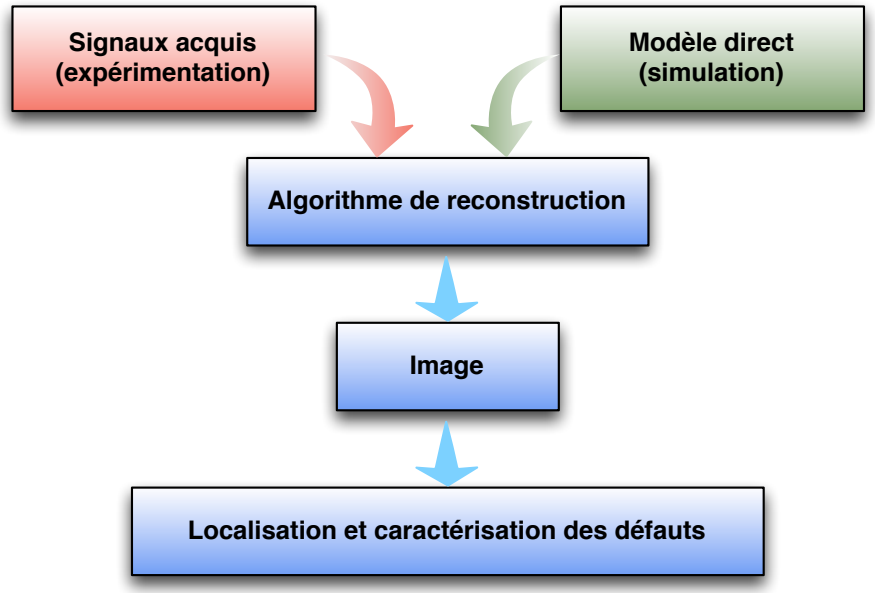


FIGURE 1.14 – Principe général des méthodes de reconstruction.

Le DISC propose deux grandes familles d'algorithmes :

- La première est basée sur une approximation du parcours de l'onde dans la pièce sous forme de polylignes appelées "trajets". Par hypothèse, on considère donc que la propagation de l'écho ultrason se fait le long d'un rayon. Ce rayon est obtenu en effectuant un lancer de rayon depuis le centre de l'émetteur avec la direction principale d'émission (généralement la normale de l'émetteur) [Porre et al., 2005]. Le rayon est propagé selon les propriétés des matériaux (calcul de réfraction aux interfaces) et sa propagation est stoppée par un critère d'arrêt (nombre de rebonds, ou sortie de la pièce). Une hypothèse importante pour ce type d'approximation est de considérer que l'onde parcourt le même trajet en émission et en réception. La figure 1.15 présente le parcours d'un trajet dans une soudure.

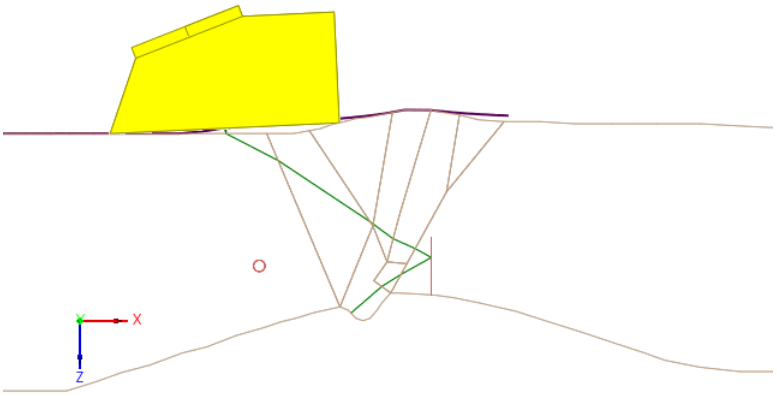


FIGURE 1.15 – Parcours d'un trajet dans une soudure.

Une fois le trajet calculé, il est possible de projeter l'information temporelle du signal dans le repère de la pièce, en tenant compte des temps de vol et des vitesses de propagation dans les matériaux. Les images reconstruites de cette manière sont appelées "vues vraies". La figure 1.16 présente la différence entre un B-Scan et un B-Scan Vrai.

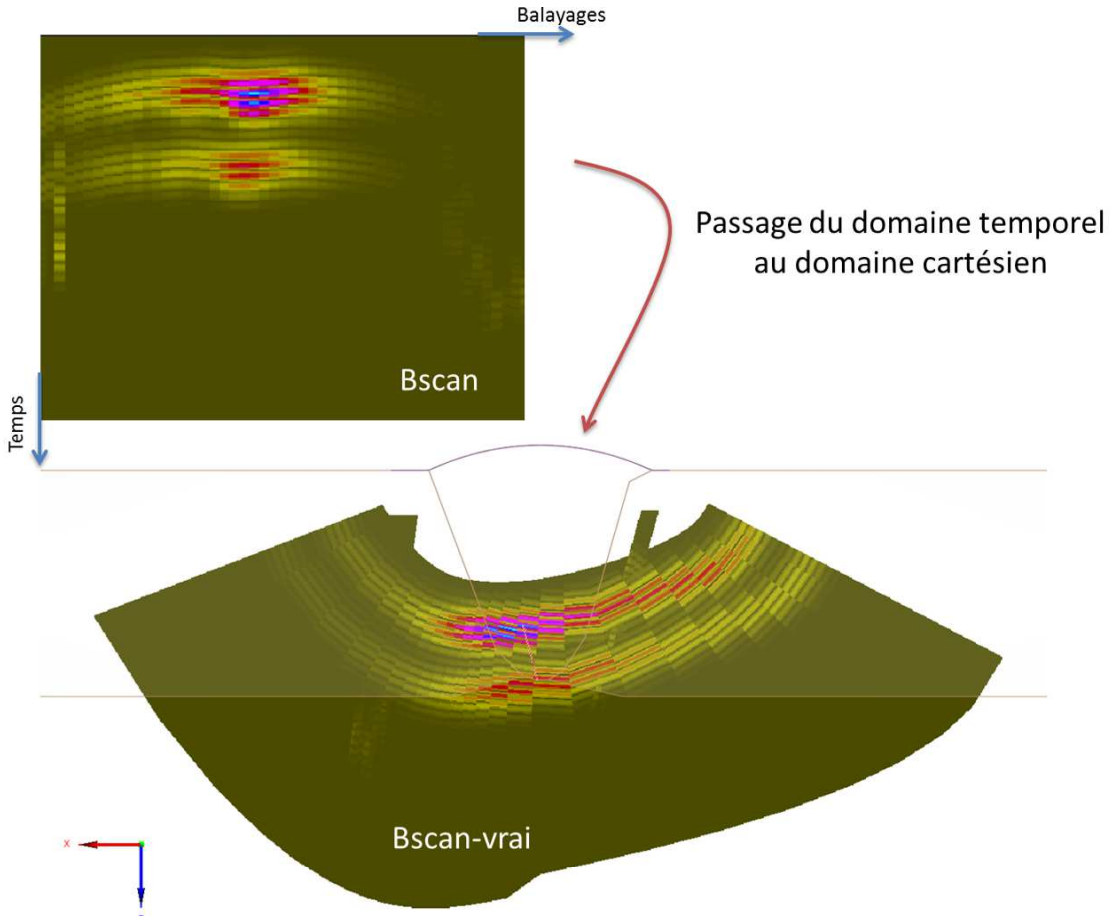


FIGURE 1.16 – Passage du domaine temporel au domaine cartésien pour un B-scan.

- La deuxième famille d'algorithmes se base sur une méthode dite par ouverture synthétique, inspirée de la méthode SAFT : la méthode de Focalisation en Tout Point (FTP) [Seydel, 1982] [Holmes et al., 2004]. Contrairement à la méthode basée sur la notion de trajet où le parcours géométrique de l'onde est utilisé, c'est le temps de propagation qui va être utilisé pour effectuer ces reconstructions. Le calcul des temps de vol de l'onde acoustique en chaque point de la zone de reconstruction peut se faire par une simulation complète du champs ultrasonore ou par une approximation géométrique. La méthode FTP fera l'objet d'une étude approfondie dans le chapitre 3 où nous nous intéresserons à la fois à la reconstruction et au calcul de simulation.

Dans le domaine du CND US, peu d'études d'accélérations d'algorithmes sur architectures parallèles sont présentes. Parmi les études existantes, on peut noter plus particulièrement deux études : l'une d'un portage d'une variante de l'algorithme FTP sur architecture GPU, et l'autre d'un portage d'un algorithme simplifié de calcul de champs ultrasonore, toutes deux réalisées par la même équipe [Romero-Laorden et al., 2011a] [Romero-Laorden et al., 2011b].

En dehors des ultrasons, on peut trouver quelques études en Courants de Foucault ou en Radiographie X. Pour cette dernière technique, on trouve par exemple un retour d'expérience présentant une nouvelle méthode de reconstruction ainsi que des gains significatifs apportés par la parallélisation sur GPU [Wang et al., 2011]. On peut aussi trouver des études plus larges, comparant les performances de multicoeurs, de GPU et de FPGA [Gac et al., 2008]. Les auteurs de cet article détaillent le travail effectué pour contourner le goulet d'étranglement existant au niveau de la bande passante mémoire sur un algorithme de rétro-projection en tomographie par émission de positrons. Un travail de parallélisation, pipeline et de pré-extraction est réalisé sur un algorithme de cache de données déjà existant. Les optimisations portent à la fois sur du réordonnancement de boucle afin d'optimiser les accès aux caches des multicoeurs et à l'usage de localité spatiale pour minimiser les besoins en bande passante.

1.1.4 La plateforme logicielle CIVA

CIVA est une plateforme logicielle qui a pour but de capitaliser les résultats de la recherche et des développements menés par le CEA et ses partenaires dans le domaine du CND. Il concerne les principales techniques que sont les ultrasons, les courants de Foucault et la radiographie X ou Gamma. CIVA propose dans un même environnement différents modules de simulation adaptés aux besoins des industriels et de recherche et développement. Son système d'imagerie ultrasonore associé à des modules de traitement d'image et du signal permet l'interprétation et l'expertise des résultats expérimentaux et de simulation.

1.1.4.1 Présentation générale

Les simulations accessibles par le logiciel CIVA permettent d'optimiser les choix d'une procédure de contrôle. En effet, le logiciel permet de prendre en compte la plupart des paramètres influents mis en jeu lors d'un contrôle, tant pour le capteur, la géométrie ou le matériau de la pièce inspectée, que pour les défauts recherchés. En jouant sur l'ensemble de ces paramètres, il est alors possible de sélectionner la méthode la plus appropriée au contrôle à effectuer, ou encore d'évaluer la performance et la pertinence d'une procédure existante.

La simulation est un atout majeur de réduction des coûts. La possibilité de faire varier les paramètres d'un contrôle (de façon maîtrisée) permet par exemple d'anticiper le contrôle dès

le stade de la conception d'un composant, ou encore de minimiser le nombre de maquettes à réaliser pour démontrer la performance d'un contrôle. La conception d'un capteur peut par ailleurs être guidée par les outils de simulation, notamment lorsqu'il s'agit de technologie multiéléments.

Les domaines d'application de CIVA sont très variés. La plateforme est utilisée dans le domaine nucléaire par la plupart des acteurs mondiaux, de la France à la Corée du Sud, en passant par les Etats-Unis, la Chine ou le Japon. Conception de nouveaux contrôles, mise en place de solutions innovantes à l'aide de capteurs multi éléments, démonstration de performance ou qualification de dossier, CIVA est un allié incontournable pour allier optimisation technique et réduction des coûts. L'aéronautique est un domaine dans lequel CIVA apporte aussi des solutions rentables. La prise en compte de matériaux complexes, la possibilité de prédire le comportement de la méthode sur des pièces de géométries variées, l'étendue considérable des capteurs et sondes disponibles donnent aux bureaux d'études et aux concepteurs de contrôles un atout qui devient rapidement incontournable.

Dans les domaines comme les transports, la métallurgie, l'aérospatiale, la pétrochimie, CIVA apporte non seulement une solution technique permettant de comprendre les phénomènes complexes, mais CIVA est aussi et avant tout un moyen de réduire les coûts. CIVA est aujourd'hui une référence dans le domaine du CND et dispose aujourd'hui de plus de 250 licences dans le monde. La plateforme est développée par 4 laboratoires du CEA-LIST et dispose de plus de 30 développeurs permanents.

1.1.4.2 CIVA ultrasons

Deux principaux types de simulation sont disponibles : le calcul de champs et le calcul de repose défaut ou calcul d'écho. Le premier permet de simuler le faisceau ultrasonore rayonné dans la pièce et éventuellement dans le couplant. Le faisceau peut être visualisé dans la matière en amplitude de couleur ou en surface iso-amplitude. Les directions locales et les fronts d'ondes peuvent également être visualisés (et sauvegardés sous forme d'animation). La figure 1.17 présente une capture d'écran issue de CIVA du point de vue utilisateur pour les simulations de champs. Le second module permet de simuler l'interaction faisceau/défaut et prédit l'amplitude et le temps de vol des échos de chaque type : direct, écho de coin. Il calcule également les échos de géométrie, l'écho de surface et tient compte des conversions de modes. C'est sur ce type d'informations que les reconstructions qui vont être étudiées dans le manuscrit sont effectuées.

Le module ultrasons prend en charge un très large panel d'éléments. Il permet de simuler l'ensemble d'une procédure de contrôle. Les différents capteurs mono-éléments et multiéléments vus en section 1.1.2.3 sont donc tous pris en charge et le choix est bien plus large puisqu'il est possible de créer ses traducteurs.

Un grand nombre de traitements du signal classiques (filtres, méthode de déconvolution, etc.), ou évolués (ondelettes, déconvolution double Bernouilli-gaussienne), sont mis à disposition. Un outil de segmentation permet le regroupement 3D des signaux, la gestion de ces groupes et la création de rapports d'examen.

Des outils de reconstruction sont intégrés au logiciel, en particulier la Focalisation en Tout Point, algorithme que nous allons étudier dans le chapitre 3. Ce traitement permet, à l'issue d'une acquisition ou d'une simulation multiéléments, de reconstruire une image présentant en

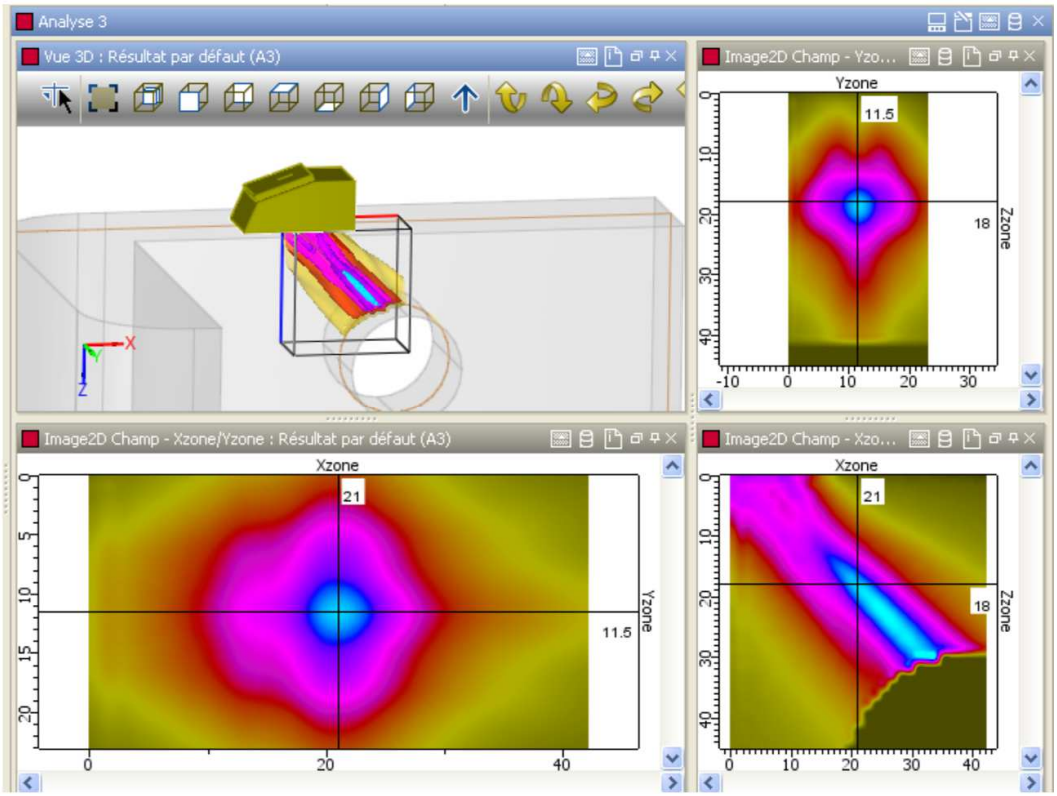


FIGURE 1.17 – Vue du module ultrasons de CIVA pour la simulation de champs ultrasonores.

chaque point l'amplitude obtenue en combinant les signaux de manière à focaliser au mieux. Un utilitaire de tracé de rayon complète ces différents outils (il gère les conversions de mode, les rebonds, et permet l'affichage des temps de parcours).

1.2 Architectures parallèles

Le Calcul Haute Performance (HPC) a longtemps été associé aux supercalculateurs, assemblages de milliers voir de centaines de milliers de processeurs. Aujourd'hui, on trouve de plus en plus d'architectures parallèles dans des machines de type station de travail et permettant d'atteindre des performances de calcul très élevées. Une carte graphique vendue 300 euros au grand public en 2009 pouvait atteindre les performances du plus puissant supercalculateur de l'an 2000, ASCI Red, avec une consommation environ 4000 fois inférieure. L'évolution est extrêmement rapide et permet d'envisager des solutions qui ne l'étaient pas quelques années auparavant. Là où le HPC se limitait à un nombre restreint de domaines tels que le nucléaire ou la météorologie, il répond maintenant à des besoins toujours plus étendus. Le marché propose des puces hautement parallèles, mélangeant processeurs généralistes (GPP) et processeurs spécialisés (e.g. les GPU).

La structure de ces nouvelles architectures a donc évolué. La montée en fréquence des GPP ayant atteint un palier du fait des difficultés à contrôler la dissipation thermique à un certain

niveau de finesse de gravure, les constructeurs ont trouvé d'autres moyens de poursuivre l'augmentation de performances. La loi de Moore, qui se vérifie encore aujourd'hui, nous indique que les puces disposent d'un nombre de transistors toujours plus importants. Un Core 2 Duo, de 2006, dispose d'environ 300 millions de transistors pour une dimension de 200 mm^2. En comparaison, un GPP Intel Sandy Bridge de 6 coeurs dispose d'environ 2,3 milliards de transistors pour une surface de 434 mm^2. En termes de performances brutes, le Core 2 Duo était capable de près de 20 GFLOPS contre environ 180 sur le 6 coeurs Sandy Bridge. Cette évolution des performances est permise grâce à l'évolution de la taille des registres SIMD qui permet d'améliorer les performances pour du parallélisme de données, ainsi qu'à la multiplication des coeurs, qui même s'ils sont plus orientés parallélisme de tâches, apportent eux aussi un moyen d'augmenter considérablement les performances.

Hormis les architectures GPP, on constate un autre courant d'évolution avec l'arrivée d'architectures novatrices telles que : les architectures multicoeurs de Tilera ou le multicoeurs 64 coeurs d'Adapteva. Le Xeon Phi d'Intel se place à la croisée entre ces deux courants d'évolution. Dans les deux cas, les constructeurs font face à un réel problème : l'utilisation du potentiel de leurs puces. Cette difficulté se répartit entre pilotes, API, compilateurs, outils de debogage et de *profiling* et modèles de parallélisme. Ces aspects sont critiques pour l'avenir des architectures.

Dans le cadre de l'intégration dans la plateforme CIVA, certaines architectures sont à exclure du fait de la complexité de mise en place sur des machines de type station de travail. Nous allons donc nous concentrer sur les architectures parallèles les plus adaptées actuellement pour une intégration industrielle, ainsi que sur les modèles pouvant être exploités sur celles-ci.

1.2.1 Les architectures GPP multicoeurs

Les deux principaux acteurs du marché des GPP multicoeurs pour machines de type station de travail sont Intel et AMD ; ARM ne ciblant pas encore ce type de machines. Ces deux concepteurs de puces ont tous deux commencé à proposer des puces intégrant du parallélisme à partir de 2006 (IBM proposait déjà un dual-core depuis 2004 avec le POWER 5, mais c'est bien Intel et AMD qui ont vraiment lancé le mouvement). Depuis, le nombre de coeurs ne cesse d'augmenter avec en 2012 des processeurs atteignant 12 coeurs pour AMD et 10 coeurs pour Intel.

1.2.1.1 Origine des multicoeurs

Ces architectures sont directement issues des processeurs monocoeurs existant auparavant. Ces anciennes architectures avaient déjà profité de nombreuses avancées à différents niveaux :

- réduction de la finesse de gravure,
- augmentation de la fréquence des transistors,
- *pipelining*,
- amélioration des mémoires caches,
- prédiction de branchements,
- exécution superscalaire.

En règle générale, ces évolutions ont toujours été dans le sens de l'amélioration d'un flux unique de calcul, et ce, aussi bien au niveau des compilateurs que des langages de programmation. Les GPP sont donc des architectures très optimisées et disposent d'un environnement de développement très complet. Les compilateurs ont évolué de manière à suivre les avancées matérielles. Les caches permettent à la plupart des développeurs d'utiliser au mieux l'architecture sans avoir nécessairement à la connaitre parfaitement. La complexité est cachée au maximum pour les utilisateurs.

Ce n'est donc que depuis quelques années que les GPP ont évolué vers des architectures parallèles. Cependant, la majorité des améliorations apportées continue d'aller dans le sens de l'exécution de flux séquentiels : il suffit d'observer la place que prennent les unités de calcul en entiers et flottants pour comprendre ce fait. En effet, elles ne prennent qu'une partie minime de la puce sur les architectures récentes.

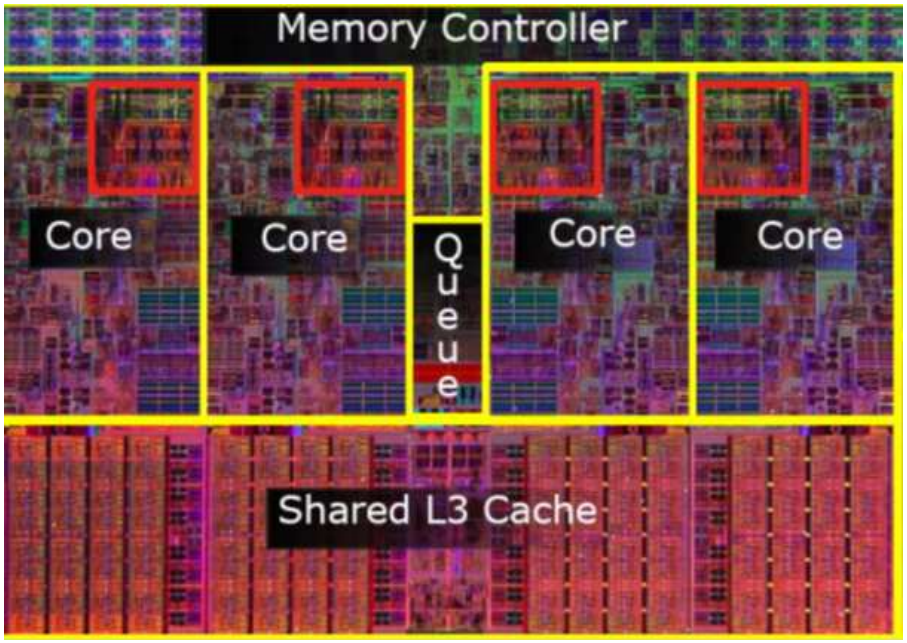


FIGURE 1.18 – Photo d'un Intel Core i7 Bloomfield de génération Nehalem disposant de 4 coeurs SMT (*Simultaneous Multithreading* ou *Hyperthreading* chez Intel), 8 Mo de cache L3. En rouge, la surface de la puce dédiée aux unités de calcul (entiers et flottants).

La figure 1.18 présente une photo de la puce d'un Intel Core i7 Bloomfield de génération Nehalem. On peut constater en rouge le peu de place prise sur la puce dédiée aux unités de calcul, ce qui illustre les propos précédents. En revanche, la place prise par le cache est très importante (environ 50% de la surface). Ces caches se basent sur le fait que les données vont être réutilisées très rapidement. Autant cela permet d'accélérer certains types de flux de calculs, autant le traitement de volumes de données importants ne va pas forcément être approprié sur ce type d'architecture. Une partie du cache pourrait donc par exemple être remplacée par des unités de calcul.

De la même manière, des unités de prédiction de branchements et des caches rapides vont permettre d'accélérer des codes où la localité des données et le nombre de branches sont im-

portants. En revanche, dans le cas de calcul intensif, comme peuvent l'être un certain nombre de codes de calcul scientifique, certaines fonctionnalités ne seront pas utilisées, rendant le GPP moins efficace.

1.2.1.2 L'architecture Nehalem

L'architecture Nehalem (Westmere) étant au centre de nos travaux, nous allons en faire une synthèse.Cette architecture comprend un ensemble de fonctionnalités à la pointe de leur évolution : exécution *out-of-order*, SIMD 128 bits, prédiction de branchements, *Simultaneous Multi-threading*, contrôleur mémoire embarqué, très grands caches dont un L3 partagé entre tous les coeurs, et un bus d'interconnexion pour relier plusieurs processeurs.

Chaque coeur dispose de quatre décodeurs d'instructions qui peuvent travailler en parallèle, même si un seul est capable de décoder les instructions x86 complexes. Les opérations sont ensuite exécutées de manière *out-of-order*. Le Nehalem dispose de registres SIMD 128 bits et de la possibilité d'exécuter jusqu'à 4 instructions par cycle d'horloge. La puissance de calcul d'un processeur Nehalem va donc être calculée de la manière suivante :

$$< Freéquence > \times < Nombre\ de\ coeurs > \times < Nombre\ d'instructions\ par\ cycle >$$

Pour un Intel Xeon X5690 Westmere disposant de 6 coeurs et fonctionnant à 3,47 GHz, cela fait donc 83 GFLOPS simple précision et moitié moins en double précision.

Quant à la bande passante, elle atteint 32 Go/s. Ce chiffre est intéressant car cela représente à peu près un accès mémoire d'un octet en DRAM contre trois opérations flottantes. Un grand nombre d'applications de calcul intensif risquent d'être limitées par la bande passante dans la mesure où théoriquement, il sera nécessaire d'effectuer 24 opérations flottantes pour une valeur double précision (8 octets).

La génération qui suit l'architecture Nehalem/Westmere est l'architecture Sandy Bridge/Ivy Bridge. Plusieurs points ont évolué, parmi lesquels :

- Latence des caches L1 et L2 a été légèrement modifiée,
- bus SIMD est doublé avec un nouveau jeu d'instructions : AVX,
- ajout d'un processeur graphique intégré partageant le cache L3 du GPP.

Les GPP continuent d'évoluer de manière importante. Pendant ce temps, leurs défauts ont tenté d'être contournés par les concepteurs d'architectures. Intel travaille depuis plusieurs années sur une puce qui initialement devait se placer sur le marché des GPU haut de gamme : le Xeon Phi (anciennement appelé Larabee, Knight's Corner, Knight's Ferry ou encore MIC). Cette puce vient d'être lancée officiellement ; Intel tente de combiner les avantages des GPP tout en augmentant au maximum le parallélisme de données avec un bus SIMD de 512 bits et de nouvelles instructions de *gather/scatter* permettant un adressage dispersé, un nombre de coeurs élevé ($\simeq 60$ coeurs). C'est du coté des puces graphiques qu'à émergé une architecture qui semble vouloir tenir dans le temps : le G80 de Nvidia.

1.2.2 Les architectures GPU

Nous commencerons par effectuer un bref historique sur les GPU puis nous présenterons l'architecture GPU Nvidia qui est au centre des travaux présentés dans ce manuscrit : l'architecture Nvidia Fermi.

1.2.2.1 Historique

Les GPU tels que nous les connaissons aujourd'hui sont relativement récents. Entre 1994 et 2001, les puces graphiques sont passées de fonctionnalités simples tel que du remplissage de pixels, à des fonctionnalités complexes telle que l'implémentation d'un *pipeline* 3D complet : transformations, éclairage, *rasterization*, applications de textures, calcul de profondeur et affichage.

Initialement, les GPU n'étaient pas programmables. En 2001, Nvidia propose, via un *pixel shader*, un premier niveau de programmabilité sur ses GeForce 3. Les possibilités sont limitées, mais l'évolution de cette architecture va apporter en souplesse de programmation. Le *pipeline* va même disposer de plusieurs étapes programmables tels que les *vertex shaders* et les *geometry shaders*.

Le GPGPU, ou *General-purpose computing on graphics processing nuits*, est réellement né durant cette période avec l'utilisation détournée des shaders à des fins de calcul généraliste. L'architecture n'étant pas adaptée, ce type de code était peu évident à développer et à maintenir. Cependant, un certain nombre de publications relatent de ces débuts du GPGPU.

En 2006, Nvidia a décidé de réunir les différents *shaders* en un seul *unified shader architecture*. Cette architecture a été développée pour le G80, pour les GeForce 8800 qui disposaient à l'époque de 128 unités de calcul distribuées en 8 *shader codes*. Chaque coeur pouvait recevoir des tâches, supprimant le besoin de gérer les différentes étapes des anciens *shaders*. En parallèle, Nvidia a aussi mis à disposition des développeurs un environnement de développement pour programmer ses GPU sans passer par le *pipeline* graphique : CUDA (pour *Compute Unified Device Architecture*). Avec CUDA, Nvidia proposait une approche bien plus aisée que la programmation des *shaders*. Nvidia a donc profité de l'élan apporté sur cette première génération programmable pour sortir une gamme de produits dédiés au calcul, du nom de Tesla. La Tesla C1060 reprenait alors les spécifications du haut de gamme du G80. Une seconde génération, le GT200 apportait un an et demi plus tard de nouvelles fonctionnalités, et une simplification des règles régissant les accès à la mémoire (DRAM et mémoires locales), et ajoutait un outil de *profiling*.

En parallèle, AMD (encore ATI à ce moment là), tentait de suivre Nvidia dans son sillage avec CTM (*Close To Metal*), une API bas-niveau qui n'aura vu le jour que sous forme de version beta. Après une annonce fin 2006, c'est donc fin 2008 que les cartes ATI devenait enfin programmables. Au travers de l'utilisation de plusieurs langages, ATI proposait donc une solution, moins aboutie que celle de Nvidia. Finalement, ATI devenu AMD, c'est OpenCL qui est adopté pour unifier l'environnement de développement des GPU AMD, début 2010.

Un an plus tôt, en 2009, alors que la communauté scientifique commence à exploiter les GPU, principalement au travers de CUDA, Nvidia sort une nouvelle génération de puce : Fermi. Nous allons présenter l'architecture GPU au travers de la présentation de cette puce, dans la mesure où les travaux de cette thèse s'appuient très fortement sur cette architecture.

1.2.2.2 L'architecture Fermi

Cette architecture, comme les précédentes d'ailleurs, a fait l'objet d'un grand nombre de présentations depuis son arrivée sur le marché. La première source d'information vient de chez Nvidia qui a publié un livre blanc présentant les spécificités de l'architecture Fermi [Nvidia, 2009]. Dans la mesure où cette architecture est au centre de nos travaux, nous allons en effectuer une synthèse de manière à mieux pouvoir faire référence à l'architecture dans le manuscrit.

FIGURE 1.19 – Schéma haut-niveau d'un GPU Nvidia Fermi.

Contrairement aux GPP, les GPU ont été créés pour effectuer des tâches relativement simples et répétitives sur un gros volume de données et sont donc plus adaptés à du calcul intensif qu'à des applications complexes et nécessitant une partie de contrôle de flux de données importante. La figure 1.19 présente un schéma haut-niveau de l'architecture Fermi. On peut constater la place que prennent les unités de calcul par rapport au cache L2. La tendance est complètement inversée par rapport au GPP, comme on a pu le voir sur le figure 1.18. Sur cette figure, on peut voir les 16 *Streaming Multiprocessors* (SM) qui composent l'architecture Fermi.

Le **Streaming Multiprocessor** (SM) est composé de 32 unités de calcul, chacune pouvant exécuter une opération flottante (ou entière) par cycle d'horloge, de même que 16 unités de *load-store* pour les accès à la mémoire, 4 unités *Special Function Unit* (SFU) exécutant les opérations complexes (e.g. trigonométrie) ainsi que 64 Ko de mémoire locale répartie entre un cache et une mémoire locale. La figure 1.20 présente un schéma d'un SM regroupant ces informations.

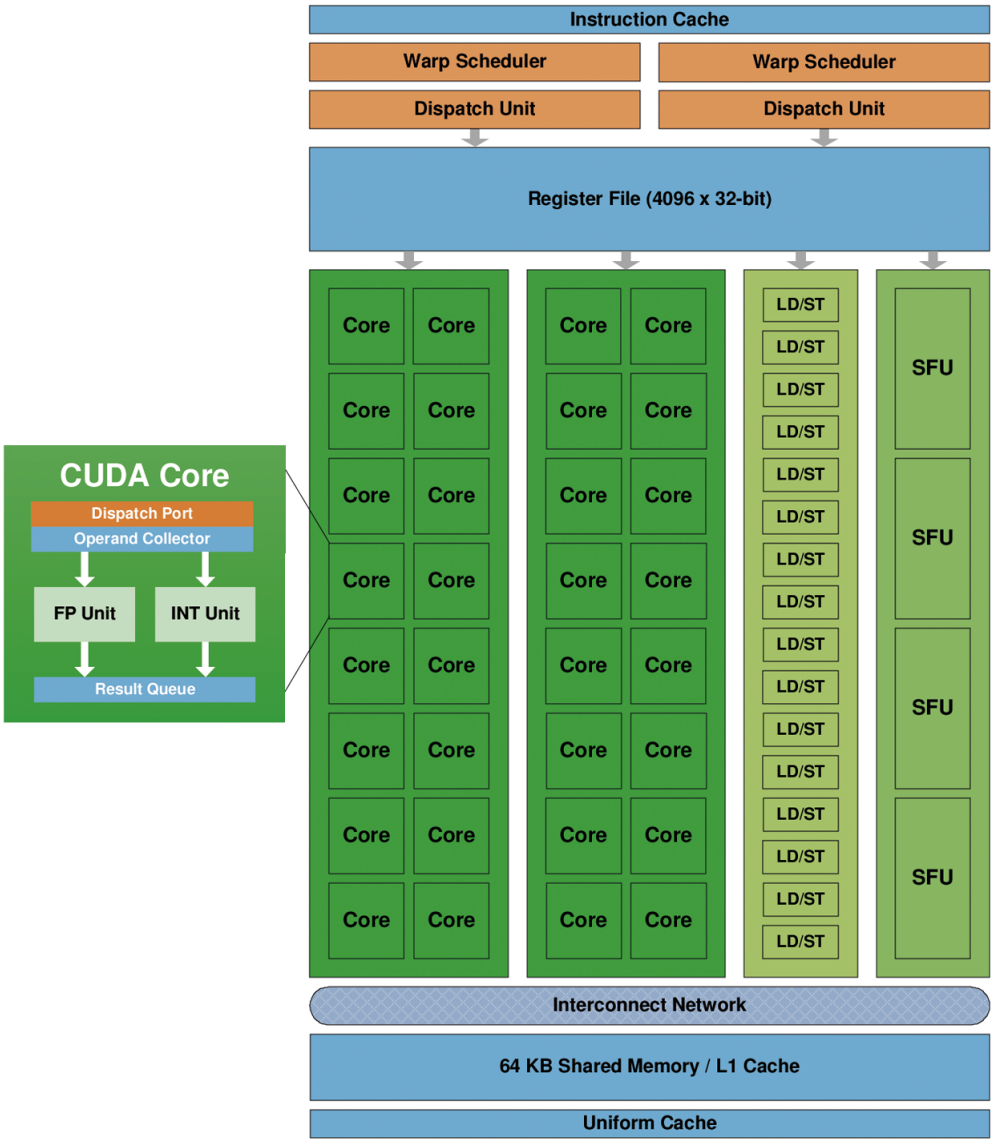


FIGURE 1.20 – Schéma d'un *Streaming Multiprocessor* d'un GPU Nvidia Fermi.

Les opérations flottantes suivent le standard IEEE 754-2008 sur les calculs flottants. Chaque coeur peut effectuer une instruction de type *fused multiply-add* simple précision en 1 cycle d'horloge et 2 double précision en 2 cycles. Fermi est à ce niveau huit fois plus performant que la génération précédente, le GT200, où la double précision et les performances d'autres opérations mathématiques telles que la division, la racine carrée et d'autres fonctions complexes étaient inférieures. Du côté de l'ALU, elle supporte les opérations classiques en 32 et 64 bits. Le support des flottants IEEE inclut les quatre modes d'arrondis et les nombres dénormalisés sont pris en charge par l'architecture de manière matérielle.

Les transactions mémoires sont gérées par les 16 unités dédiées dans chaque SM. Fermi permet aussi de traiter des accès "2D". Les conversions de types sont gérées à la volée lors du passage de la mémoire globale aux registres ou inversement.

Les SFU exécutent les opérations telles que *sin*, *cos*, et *exp*. Quatre opérations par cycle peuvent être exécutées par ces unités.

Un SM est en réalité séparé en deux parties de 16 coeurs, indépendamment des SFU et des unités dédiées aux accès à la mémoire. Les deux ordonnanceurs dont dispose le SM vont donc pouvoir gérer l'ordonnancement de manière à recouvrir au mieux les importantes latences à la mémoire globale (400 à 800 cycles mémoire). Un système de "burst" permet lorsqu'un accès est réalisé, de récupérer un segment mémoire (de 32 à 128 octets), de sorte que lorsque les threads sont bien ordonnancés, de ne payer que le coût d'un accès mémoire pour un groupe de 16 ou 32 threads.

La *hiérarchie mémoire* des GPU est soumise à des contraintes assez fortes. Comme pour la génération précédente, chaque SM dispose d'une mémoire locale dite *shared* de 64 Ko. La différence par rapport au GT200 est que cette mémoire est partagée en deux espaces, l'un de mémoire cache L1, et l'autre de mémoire locale gérable manuellement par l'utilisateur. Deux configurations sont possibles avec une répartition 16/48 Ko ou 48/16 selon le besoin. La mémoire *shared* dispose de latences d'accès très faibles, mais est relativement limitée en taille. Elle est donc utilisée pour stocker des résultats intermédiaires avant de les ressortir en mémoire globale. Cette mémoire est locale à un SM et est donc privée vis-à-vis d'une vue globale du GPU. Le choix entre les deux modes peut se faire à l'exécution.

Les SM disposent ensuite d'un cache L2 partagé de 768 Ko. Ce cache a une particularité par rapport aux GPP actuels., il est capable au travers des contrôleurs mémoire des SM de gérer des instructions atomiques. Ces instructions permettent une lecture, modification ou écriture de manière séquentielle, soit sans être interrompue par un autre thread utilisant la même donnée. Ce type d'opération a évidemment un coût, mais est aussi beaucoup plus simple d'utilisation que la mise en place d'un mutex ou d'un sémaphore pour synchroniser l'accès à des données sur les GPP. Les instructions atomiques peuvent bloquer l'accès à une adresse mémoire spécifique, tout en permettant au reste de la mémoire de fonctionner de manière classique. D'après Nvidia, ces instructions bénéficient d'instructions de 5 à 20 fois plus performantes que celles de la génération précédente.

Le dernier niveau de mémoire est la DRAM, ou mémoire globale (GDDR5). Fermi peut prendre en charge jusqu'à 6 Go sur les Tesla C2070. Cette mémoire gère l'ECC (*Error Correcting Code*), et le système existe aussi au niveau des caches et des registres. Cette protection est particulièrement intéressante pour les calculs de très longue durée.

Fermi a apporté de nombreuses modifications principalement dédiées au calcul. Tout en améliorant les performances graphiques et de calcul, Nvidia a réussi le pari d'améliorer aussi son architecture en termes de souplesse de programmation grâce aux instructions atomiques et à l'assouplissement des règles d'accès à la mémoire globale.

1.3 Outils de programmation et de parallélisation

Les architectures présentées ci-dessus disposent d'outils de programmation natifs. Les GPU Nvidia ont CUDA, tandis que les GPP multicoeurs disposent de plusieurs outils : OpenMP, TBB ou encore Cilk++. Nous allons présenter brièvement ces outils.

1.3.1 Outils natifs

1.3.1.1 CUDA

CUDA, ou *Compute Unified Device Architecture*, est une architecture de calcul parallèle développée par Nvidia permettant de programmer les GPU de la marque. CUDA inclut un ensemble d'instructions et un moteur de calcul parallèle sur le GPU. CUDA propose un modèle pour l'exécution de codes ainsi qu'un modèle mémoire adapté à l'architecture mémoire des GPU (cf. section 1.2.2.2).

Le modèle d'exécution
Le modèle de programmation de CUDA diffère des modèles multithreadés des GPP. Le développeur CUDA visualise sa machine sous la forme d'un hôte ou *host*, un GPP classique, et d'un ou plusieurs processeurs de calcul massivement parallèle équipés d'un très grand nombre d'unités de calculs. Le langage CUDA est directement lié à l'architecture des GPU Nvidia. Le modèle de programmation est basé sur la notion de *kernel*. Un *kernel* est une fonction qui est appelée depuis le *host* et exécutée sur le GPU par un très grand nombre de threads de calcul en parallèle. Le *host* utilise des données résident en mémoire RAM sur le GPU. Le code CUDA est basé sur du C ANSI, étendu avec un certain nombre de mots clés permettant d'identifier les kernels, et les types de données utilisés.

Néanmoins, CUDA prend aussi en charge un sous-ensemble du C++ au travers des templates, des références, de la surcharge d'opérateurs et de classes basiques. Le compilateur Nvidia, NVCC, divise le code entre la partie *host* et la partie *device*. La partie *host* est compilée comme du C ou C++ classique (avec le compilateur par défaut de la plateforme GCC/Visual/ICC). Quant à la partie *device*, elle est d'abord transformée en un langage intermédiaire (PTX), puis en assembleur GPU. En fin de chaine, les deux parties sont liées dans un exécutable unique qui contient à la fois le code GPP et le code GPU.

Le meilleur moyen d'obtenir de bonnes performances avec une telle architecture est de programmer d'une manière assez proche de celle du SIMD (*Single Instruction Multiple Data*) car le grand nombre de coeurs de calculs peut exécuter le même code de manière synchrone sur différentes données. Nvidia parle de modèle SIMT (*Single Instruction Multiple Threads*). Lors d'un appel à un *kernel*, il est exécuté sous la forme d'une grille de threads (threads très légers à ordonnancer, en comparaison aux threads GPP).

La création de ces threads est effectuée sur le GPU et est extrêmement rapide. Le temps d'instanciation a été mesuré et varie autour de $10\ \mu s$ quelle que soit la quantité de threads [Volkov and Demmel, 2008]. La meilleure manière d'obtenir de bonnes performances est donc d'exécuter une très grande quantité de threads (plusieurs milliers) qui seront exécutés en parallèle. Le code d'un *kernel* va par exemple regrouper le corps d'une boucle for en C et sera

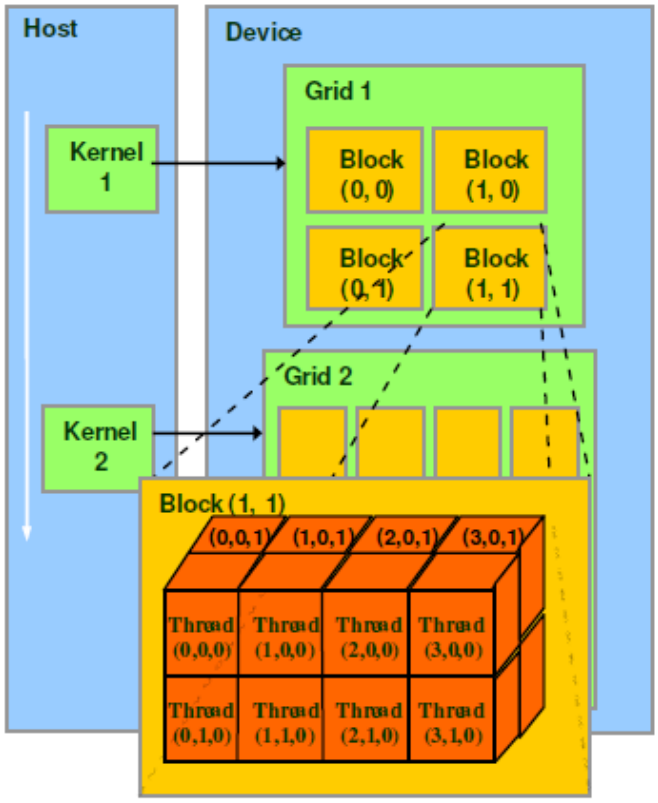


FIGURE 1.21 – Hiérarchie de groupement des threads.

exécuté de manière massivement parallèle sur le GPU au lieu de manière séquentielle.

Un *kernel* nécessite un paramétrage manuel du nombre de threads concurrents. Chaque thread qui exécute un *kernel* dispose d'un identifiant accessible via une variable. Les threads sont groupés en blocs et ces blocs sont organisés dans la grille (*grid*). Ces deux niveaux de hiérarchie sont présentés sur la figure 1.21. Cela permet d'exprimer simplement un calcul sur un domaine de type vecteur/matrice par exemple. Les threads d'un bloc peuvent coopérer entre eux au travers la mémoire *shared*.

Le modèle mémoire

En CUDA, *host* et *device* disposent d'espaces mémoire séparés, tel que présenté au niveau matériel, où les GPU disposent de leur propre DRAM. Afin d'exécuter un kernel sur le *device*, le développeur doit allouer de la mémoire sur ce *device*, puis transférer les données depuis le *host*. De la même manière, après l'exécution, le développeur doit transférer les données vers le *host* et libérer la mémoire du GPU. Ces fonctionnalités sont mises à disposition du développeur au travers de l'API CUDA.

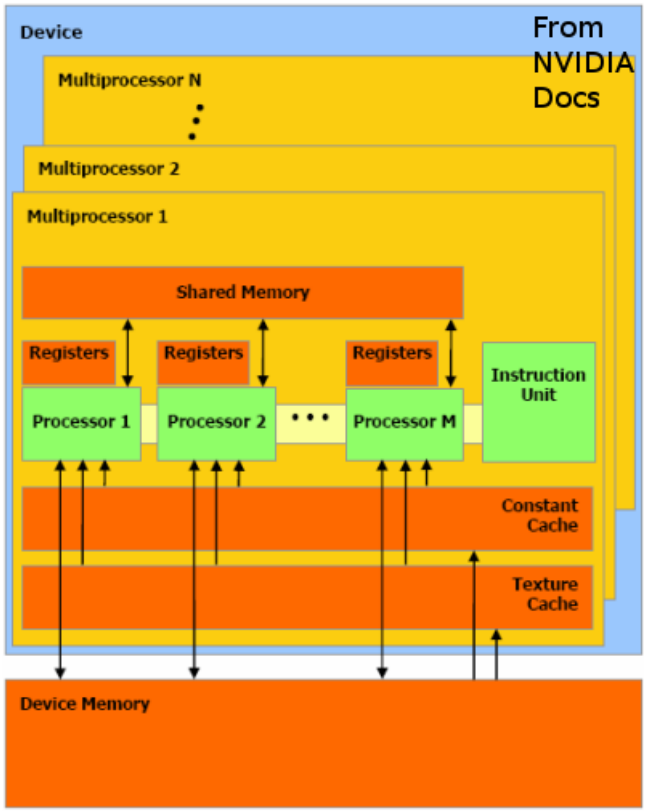


FIGURE 1.22 – Hiérarchie de groupement des threads.

La figure 1.22 présente un schéma du modèle mémoire de CUDA. Chaque thread dispose d'une mémoire privée locale. Il dispose aussi d'une mémoire *shared* visible par tous les threads de son bloc et dont la durée de vie est la même que celle du bloc. Enfin, tous les threads ont accès à une mémoire globale ou *global*, la DRAM du GPU. Les GPU disposent aussi d'espaces mémoire uniquement utilisables en lecture, que sont la mémoire *constant* et la mémoire *texture*. Ces différents espaces mémoires sont optimisés pour différentes types d'utilisation. Ces deux mémoires et la mémoire globale ont une durée de vie qui ne dépend pas des *kernels* et peuvent persister tout au long d'une application.

Il est important de noter que des différentes optimisations sont liées au matériel et donc aux bandes passantes, latences, et autres spécificités matérielles. Par exemple, la mémoire *shared* dispose de latences du même ordre que les registres du moment que les accès sont réalisés sans conflits tandis que les accès en mémoire globale sont deux ordres de grandeur plus lents (400 à 800 cycles en comparaison à une dizaine de cycles pour un accès en *shared*).

L'environnement de développement

CUDA dispose, en plus de son interface propre, d'une interface OpenCL et d'une DirectCompute (DirectX). De nombreux *wrappers* ont été développés pour Python, Java, .NET ou encore Matlab. Les outils de compilation sont disponibles sous Windows, Linux et Mac OS X. De plus, Nvidia a mis à disposition un environnement associé à Visual Studio, Nsight, permettant le *profiling* de l'application au travers de nombreux éléments d'analyse, ainsi que d'un outil de débogage. Ces outils sont précieux, dans un domaine encore jeune et là où par exemple AMD, ne dispose que d'outils beaucoup moins aboutis pour ses GPU.

La maitrise de l'architecture des GPU Nvidia et de l'environnement de développement a été principalement acquise lors d'une campagne d'implémentations et de benchmarks réalisés à l'iEF, en parallèle de ma thèse [Courbin et al., 2009] [Pédron et al., 2010].

1.3.1.2 OpenMP

OpenMP est un ensemble de modules utilisés pour bénéficier des processeurs multicoeurs. Cette bibliothèque est disponible pour les langages Fortran, C et C++ et a l'avantage d'être supportée par les systèmes Unix et Windows. L'utilisation se fait via un ensemble de directives et de variables d'environnement et d'une bibliothèque de fonctions qui permettent d'être peu intrusif au niveau du code. Cette interface de programmation est l'une des plus anciennes, ayant vu arriver ses premières spécifications pour Fortran en 1997.

Principe général

OpenMP est basé sur le fait qu'un thread va effectuer un certain nombre de *fork* pour créer des threads esclaves et partager une tâche de calcul entre eux. Ces threads sont répartis sur les différents coeurs de la machine et sont exécutés en parallèle. Une directive est utilisée pour marquer la section de code à exécuter en parallèle. Chaque thread dispose d'un identifiant qui est obtenu en utilisant la fonction `omp_get_thread_num()`, ce qui permet, entre autres, d'utiliser différents segments mémoire en fonction du numéro de thread. OpenMP permet donc, par ce biais, d'exposer à la fois parallélisme de tâche et de données. L'ordonnancement des threads est pris en charge par un ordonnanceur paramétrable comme nous allons le voir par la suite.

Les primitives mises à disposition par OpenMP permettent, entre autres, le découpage de tâches automatique, la gestion du partage des données (publiques/privées), la synchronisation entre les threads ou encore la gestion du nombre de threads à allouer. En C et C++, les directives utilisent des *pragmas*.

Allocation de threads

L'allocation se fait via la primitive suivante : `#pragma omp parallel` Cette directive se place en amont de la section de code que l'on souhaite exécuter sur plusieurs coeurs. Par défaut, OpenMP détecte le nombre de threads logiques maximum sur la machine et alloue autant de threads. Sur un processeur à quatre coeurs sans SMT, OpenMP allouera par défaut 3 threads, en plus du thread principal. Ce nombre de threads peut être passé de manière statique dans le code ou peut aussi être passé via la variable globale `OMP_NUM_THREADS`.

Bien que les capacités d'OpenMP soient nombreuses, l'utilisation principale est la parallélisation de boucles `for`. Voici une simple addition de deux matrices 2D :

```
for(int i=0; i<N; i++) {
  for(int j=0; j<M; j++) {
    b[i][j] += c[i][j];
  }
}
```

La parallélisation via OpenMP va simplement se faire de la manière suivante :

```
#pragma omp parallel for
for(int i=0; i<N; i++) {
  for(int j=0; j<M; j++) {
    b[i][j] += c[i][j];
  }
}
```

Les itérations de la boucle « for » vont être réparties entre les threads OpenMP. Le choix de placer le *pragma* sur la boucle la plus à l'extérieur est qu'il n'y a aucune dépendance de données entre les itérations des deux niveaux de boucles. Ainsi, les threads vont avoir plus de calculs à effectuer et donc minimiser les retours vers l'ordonnanceur. Cet ordonnanceur dispose de trois modes de fonctionnement :

- `static` : le découpage des itérations est régulier et la répartition est préparée à la compilation
- `dynamic` : ce système permet, pour des itérations moins régulières de réguler la charge des threads à l'exécution. Ce mode peut être intéressant lorsque les itérations divergent en quantité de calcul, mais nécessite de nombreux allers retours sur le thread principal ce qui peut s'avérer pénalisant.
- `guided` : solution intermédiaire qui permet d'avoir une partie du découpage à la compilation et une autre à l'exécution.

A ces modes d'ordonnancement s'ajoutent un paramètre qui donne la taille minimale d'une sous-tâche. Cela permet de grouper certains itérations et par exemple de bénéficier des caches L1 et L2 de manière plus importante.

Autre possibilité, OpenMP permet d'effectuer des réductions. Voici un code qui calcule la somme des valeurs d'un tableau, initialement sans OpenMP, puis avec lui :

```
// Boucle séquentielle
for(i = 0; i < N; i++){
   sum = sum + b[i];
}

// Boucle parallélisée
#pragma omp parallel for shared(sum)
for(i = 0; i < N; i++){
   val = b[i];
   #pragma omp critical
   sum = sum + val;
}
```

L'utilisation du mot clé `shared` est donc faite pour la variable partagée. De plus, de manière à ce que les threads évitent les collisions sur la somme, il est nécessaire d'utiliser une directive `#pragma omp critical` pour sérialiser l'accès à cette variable partagée. Une solution plus simple existe via une clause qui permet directement d'effectuer la réduction :

```
#pragma omp parallel for private(sum) reduction(+: sum)
for(i = 0; i <N; i++){
   sum = sum + b[i];
}
```

De cette manière l'accès à la variable `sum` est sérialisé.

Le point important est qu'OpenMP permet de paralléliser des boucles `for` de manière relativement aisée. Le paramétrage de l'ordonnanceur permet de gérer les différents cas qui se présentent, tel qu'un mauvais équilibrage des calculs des itérations de boucles. Le système par directives est peu intrusif, même si la plupart du temps, il est nécessaire de modifier certaines allocations mémoires en vue de la parallélisation.

1.3.1.3 Threading Building Blocks

Intel Threading Building Blocks (TBB) est une bibliothèque template C++ visant, comme pour OpenMP, à bénéficier des architectures multithreadées des processeurs multi-coeurs actuels et à venir. Cette bibliothèque est plus récente qu'OpenMP, datant de l'arrivée des architectures Intel Core. TBB propose un ensemble de structures de données et d'algorithmes permettant d'éviter les écueils de la programmation multithread manuelle. Les calculs sont répartis en tâches qui sont ensuite envoyées aux différents coeurs du processeur de manière dynamique, tout en maximisant l'utilisation des caches.

TBB utilise un système de *task stealing* pour optimiser l'équilibrage de charge entre les threads. Les tâches sont partagées entre tous les threads. Dès lors qu'un thread termine sa file de calcul, si les autres threads n'ont pas terminé leur travail, l'ordonnanceur va récupérer des tâches des files des threads en calcul pour les donner au thread en attente. Ce mécanisme permet de maximiser l'efficacité du parallélisme tout en minimisant les communications avec un thread principal tel que dans OpenMP.

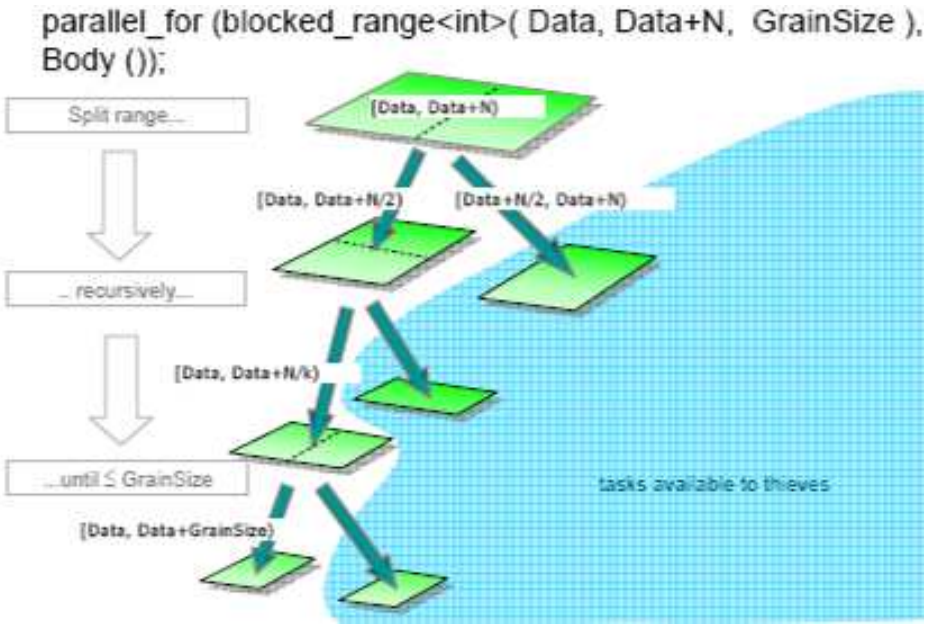


FIGURE 1.23 – Découpage des données pour la primitive TBB `parallel_for`.

TBB met à disposition des outils de parallélisation de boucles tels qu'OpenMP. Voici l'exemple d'une somme de vecteurs à l'aide de la primitive `parallel_for` (cf. figure 1.23 pour la manière dont sont partagées les données) :

```
void vectorAddInPlace(int b[], int c[], size_t n) {
  parallel_for(size_t(0), n, size_t(1) , [=](size_t i) {b[i]+=c[i];});
}
```

Ne serait-ce que sur cet exemple, on peut constater que l'utilisation d'une bibliothèque telle que TBB sera plus intrusive qu'OpenMP. Là où OpenMP est limité à des réductions sur une valeur, TBB va pouvoir proposer des réductions sur des valeurs complexes, tels que des tableaux. TBB est une bibliothèque très complète, mais nécessite en règle générale des développements plus couteux et plus complexes que ceux réalisés avec une interface telle qu'OpenMP.

1.3.1.4 Cilk+

Cilk Plus est en quelque sorte un sous-ensemble de Thread Building Blocks. L'idée derrière la création de Cilk est identique à celle de TBB : elle repose sur le fait que le développeur doit être capable d'exprimer le parallélisme dans son application de manière explicite. En revanche, Cilk se veut beaucoup plus simple. Dès lors que différents segments de codes sont indépendants et peuvent être exécutés en parallèle, ils doivent être parallélisés.

A l'origine, Cilk a été développé au MIT depuis 1994 et visait initialement le C. En juillet 2009, Intel a racheté Cilk et a développé un module C++, sous le nom de Clik++. Cette interface se base sur trois mots clés :

- `cilk_spawn` : création de threads. Permet d'indiquer qu'un code peut être exécuté en parallèle sans dépendances,
- `cilk_sync` : barrière de synchronisation. Signale à la procédure qu'elle ne peut continuer tant que les différents threads générés par les précédents spawn n'ont pas terminé leur travail,
- `cilk_for` : permet de paralléliser une boucle `for` de manière simple.

Cilk permet donc d'accéder au parallélisme de données via la primitive `cilk_for` et au parallélisme de tâches via `cilk_spawn` et `cilk_sync`.

1.3.2 Outils hybrides

Les outils de programmation permettant de cibler différents types d'architectures ont aussi été développés. Le niveau adopté par les différentes API/interfaces n'est pas toujours le même. Par exemple, d'un coté OpenCL fait le choix du bas niveau, tandis qu'un outil comme HMPP va, à l'instar d'OpenMP pour les GPP multicoeurs, chercher à aller au plus simple pour effectuer une parallélisation. Nous allons présenter brièvement les deux modèles nommés ci-dessus, ainsi qu'OpenACC, un nouveau standard émergent.

1.3.2.1 OpenCL

OpenCL est un standard ouvert et gratuit dédié à la programmation parallèle de machines hétérogènes. Il fournit aux développeurs un environnement uniformisé permettant de programmer différents types de machines telles qu'un ensemble de GPP multicoeurs, de GPU, d'architectures de type Cell ou encore d'autre processeurs comme certains DSP.

Développé par Khronos group, l'objectif principal d'OpenCL est de fournir un ensemble d'applications, depuis l'embarqué jusqu'au HPC, à l'aide d'une abstraction bas-niveau, mais tout aussi portable. Il faut savoir que le consortium qui à mis au point ces spécifications est composé des principaux acteurs du marché, à savoir : AMD, Apple, ARM, Intel, Nvidia, Sony, et plusieurs autres grands groupes. Ils espèrent qu'OpenCL se place comme fondation d'un environnement informatique dédié au calcul parallèle incluant outils, middleware et applications. C'est ce même consortium qui maintient et gère l'évolution d'OpenGL et d'OpenAL.

OpenCL s'appuie sur trois modèles : plateforme, exécution et mémoire.

Modèle de plateforme
Un hôte est connecté à un ou plusieurs dispositifs compatibles OpenCL. Chaque dispositif est composé d'un ou de plusieurs coeurs de calcul.

Modèle d'exécution

- Kernel : fonction de calcul basique avec parallélisme de données ou de tâches.
- Programme : ensemble de kernels et d'autres fonctions, qu'on peut voir sous forme de bibliothèque.
- File de kernels : exécutable de manière FIFO ou possibilité d'ordonnancement manuel.

Modèle mémoire
Comme le montre la figure 8, la structure est calquée sur les architectures des dernières générations de GPU. Plusieurs unités de calculs, elles-mêmes sous divisées en plusieurs coeurs incluant deux niveaux de mémoires internes, et un externe au niveau de l'hôte. Ce modèle est suffisamment généraliste pour s'adapter à la future puce Intel Xeon Phi. Concernant les GPP, OpenCL est pris en charge, particulièrement pour du parallélisme de tâches, mais ce n'est clairement pas l'objectif premier.

Concernant ces derniers, une volonté commune de normalisation a émergé sous la forme d'OpenCL. Son lancement a été grandement favorisé par Mac OS qui inclut l'accès à l'API OpenCL dans le but d'accélérer des tâches ponctuelles. Les principaux partenaires du consortium étant Apple, Nvidia et AMD, le modèle de programmation se prête parfaitement aux GPU. Mais le but d'OpenCL est aussi de cibler d'autres architectures que les GPU comme par exemple les GPP multicoeurs ainsi que d'éventuelles puces exotiques telles que l'Intel Xeon Phi, où la tendance est clairement à la multiplication des coeurs.

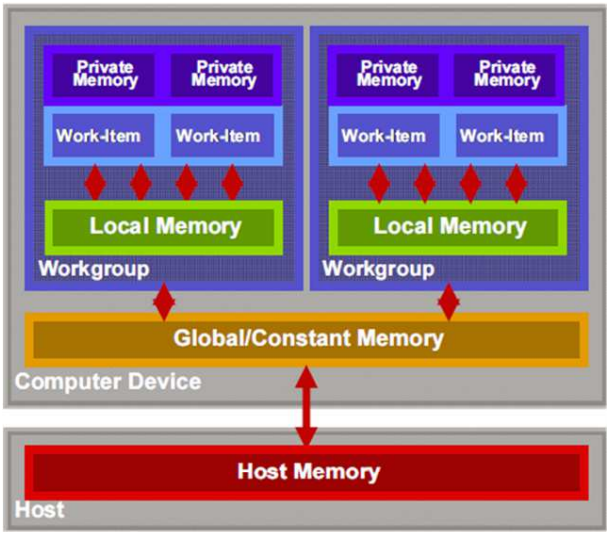


FIGURE 1.24 – Modèle mémoire adopté par OpenCL.

1.3.2.2 HMPP

L'environnement de développement Hybrid Multi-core Parallel Programming (HMPP) est développé par CAPS. Il comprend un ensemble d'outils dédiés à la parallélisation hybride pour des systèmes multicoeurs. Cela regroupe des accélérateurs de type GPU (Nvidia ou AMD) ou des multi-coeurs SIMD. L'environnement comprend un compilateur pour Fortran et C et un runtime qui permet d'utiliser les outils de développement et drivers des architectures cibles.

Interface logicielle

HMPP comprend un ensemble de directives telles que celles d'OpenMP de manière à modifier le moins possible le code source en séparant les portions de codes à accélérer de manière explicite. Les directives HMPP sont basées sur trois types :

- Codelet : permet de définir une fonction à accélérer,
- Execution : permet de spécifier l'exécution d'un codelet dans le programme,
- Data Transfers : les données peuvent être envoyées avant l'exécution d'un codelet et récupérées ensuite après l'exécution.

Une directive codelet déclare qu'un calcul va être exécuté sur un accélérateur. Le type d'architecture ciblée peut être indiqué, mais HMPP sélectionnera par défaut le premier accélérateur compatible s'il n'est pas spécifié. Afin d'optimiser le portage pour une architecture ciblée, le calcul peut être spécifié en fonction des paramètres, de leur taille, de leur valeur, etc. De plus, si l'accélérateur n'est pas disponible, le code de base peut toujours être exécuté et le comportement de l'application préservé. L'exécution d'un codelet peut être synchrone ou asynchrone. HMPP peut être vu comme couche logicielle faisant le lien entre un environnement de programmation spécifique et le calcul au sens large. HMPP peut donc être particulièrement intéressant lors de l'utilisation de kernels pour des cas spécifiques (comme l'utilisation d'un kernel spécifié uniquement pour des vecteurs de grande taille).

Environnement d'exécution

Le runtime dont dispose HMPP est une bibliothèque gérant les appels et les calculs envoyés aux accélérateurs matériels (HWA). Cette bibliothèque alloue et initialise le HWA de manière à permettre exécution de codelets. Elle gère aussi la communication entre l'hôte et le HWA et

gère les appels asynchrones.

Une application HMPP peut être compilée et exécutée avec un compilateur classique, qui ignorera les directives HMPP. Cette application est parsée par le pré-processeur pour extraire les parties contenues par les directives HMPP et les transformer en appels au runtime HMPP. L'édition de liens dynamique permet de modifier l'architecture cible ou de changer de codelet sans avoir à recompiler l'application. HMPP se place donc comme une solution permettant d'accélérer certaines portions de code.

Pendant la génération du code, le développeur peut consulter et modifier le code spécifié pour un accélérateur avant la compilation finale qui est réalisée pour une cible spécifique. Ce principe permet à un expert d'effectuer une phase d'optimisation manuelle. En théorie, cela permet de faire gagner du temps au développeur qui s'affranchit de l'encapsulation des appels, des transferts mémoire, etc.). La figure 1.25 présente un schéma des différentes étapes de compilation pour un code ciblant GPP et GPU.

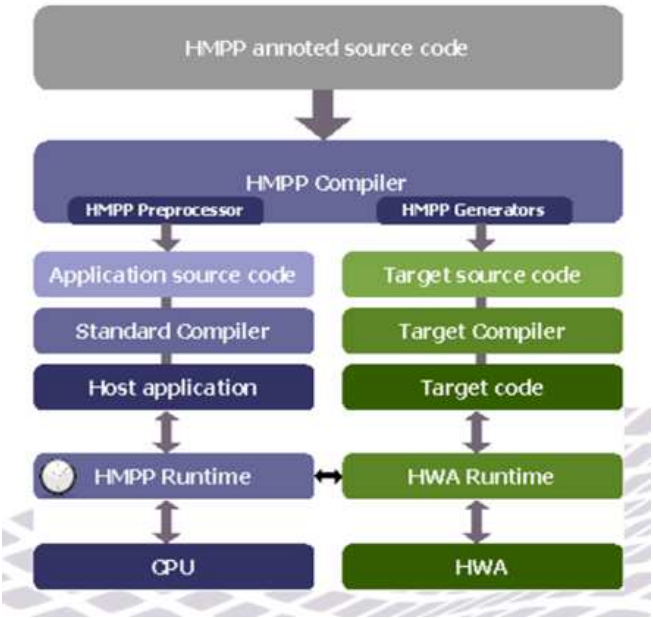


FIGURE 1.25 – Etapes de compilation d'un code HMPP.

1.3.2.3 OpenACC

OpenACC regroupe un ensemble de pragmas haut-niveau qui permettent au C, C++ et Fortran de programmer des architectures massivement parallèles avec une facilité d'utilisation telle que celle d'OpenMP. Il s'agit de l'alternative la moins intrusive à disposition. OpenACC dispose d'un langage relativement riche permettant d'indiquer l'emplacement mémoire de données, de transférer ces données, ou encore de paralléliser boucles ou sections de code. A l'instar d'OpenCL, OpenACC va permettre de sélectionner une plateforme et une architecture et de récupérer différentes informations les concernant.

Le développement d'OpenACC est soutenu par PGI (The Portland Group), au travers de leur compilateur. Actuellement le compilateur ne prend en charge que le C et Fortran, et se

limite à la plateforme GPU Nvidia. Cependant, le standard a pour but de cibler aussi bien les GPU AMD, ainsi que l'Intel Xeon Phi. PGI propose un système de profling qui permet d'utiliser, pour la plateforme Nvidia, l'outil de profiling « computeprof ».

OpenACC propose un modèle d'exécution très semblable à ceux d'OpenCL ou CUDA, avec trois niveaux : gang, worker et vector. Si l'on compare à CUDA, le gang correspond au bloc de threads, le worker, un warp, et enfin, le vector correspond au thread. Comme pour CUDA, le partage de données entre gangs n'est pas possible. Ce choix permet d'obtenir un meilleur passage à l'échelle, quel que soit le nombre d'unités de calcul. Ce modèle permet a priori de s'adapter aux architectures. Cela veut aussi dire que pour tirer le meilleur des performances d'une architecture, il sera toujours nécessaire d'adapter le code manuellement et spécifiquement pour cette architecture.

Même si le standard est encore très récent, il semble que ce type d'outils soit beaucoup plus adapté à des développements de dimensions importantes. En effet, le système de directives évite des réécritures complètes de code que va demander par exemple OpenCL.

1.3.3 Conclusion sur les modèles de programmation

Nous avons pu voir que ces différents modèles hybrides ont de nombreux points communs. Ils se différencient principalement sur le niveau de programmation. Certains modèles proposent des outils destinés à une réécriture complète d'un code, comme pour OpenCL. D'un autre côté, HMPP ou OpenACC proposent des solutions à base de directives, permettant de minimiser les modifications de code.

Peu de benchmarks comparatifs montrent les performances de ces différents modèles pour une même architecture donnée. Il est donc relativement difficile de les comparer en termes de résultats et de performances pures. Par contre, nous pouvons constater que là où OpenMP a très bien réussi, en adoptant une approche haut niveau, basée sur des directives, il est tout à fait possible que ce type d'outil soit aussi la solution la plus appropriée à la programmation hybride.

Il est à noter que ce ne sont pas les seules solutions existantes. En effet, Microsoft développe aussi sa solution propriétaire, C++AMP. Cette bibliothèque C++ utilise DirectX 11 pour cibler les architectures GPU et repose sur la base posée par la PPL (Parallel Parterns Library). Ce modèle est encore très récent et n'est pas encore très exploité. Les quelques retours d'expériences pointent le manque d'outils disponibles, ou encore un certain manque de stabilité selon le matériel utilisé, mais l'intégration dans Visual Studio 2012 semble prometteuse, le code est beaucoup moins verbeux que pour OpenCL, et il semblerait qu'il soit possible de capturer des exceptions de manière à gérer l'arrêt d'un noyau de calcul sans pour autant obtenir une erreur nécessitant le redémarrage de la machine, comme cela pouvait être le cas pour CUDA ou OpenCL à leurs débuts.

1.4 De l'intégration de codes parallélisés dans un contexte industriel

L'intégration de codes optimisés dans une plateforme industrielle est une problématique qui existe depuis déjà quelques années. En effet, que ce soit la programmation SIMD, le GPGPU, ou encore l'utilisation d'instruction assembleur, toutes ces techniques, aussi performantes soient-elles, posent certains problèmes lorsqu'il faut les intégrer dans un logiciel industriel. L'un des objectifs du manuscrit étant d'apporter des solutions pérennes à une plateforme logicielle telle que CIVA, il nous a semblé pertinent de proposer une synthèse des problèmes que l'on peut rencontrer lors de l'intégration de codes optimisés et des moyens existants de les gérer.

1.4.1 Langages de programmation

Un logiciel est bien souvent directement dépendant de l'architecture logicielle choisie à ses débuts. En effet, certains choix sont effectués sur le moment et peuvent parfois s'avérer être des erreurs, selon l'évolution des langages et de leurs fonctionnalités. La plupart des outils dédiés au parallélisme sont aujourd'hui disponibles principalement en C et C++ :

- CUDA, API C/C++,
- OpenCL, API C,
- OpenACC, langage basé sur des primitives, peut cibler C et Fortran,
- OpenMP, disponible en C, C++ et Fotran.

Si l'application que l'on cherche à accélérer n'est pas développée dans un de ces langages, ou que le langage n'est pas supporté par le type d'API souhaité, il est nécessaire de s'orienter vers des wrappers. Pour CUDA par exemple, il existe un grand nombre de wrappers pour différents langages : Python, Java, Matlab, IDL, C#. Même si ces wrappers ont l'avantage d'exister, ils posent évidemment un problème de maintenance. En effet, ces wrappers sont très rarement officiels, le support est donc très limité. Dans le cas d'utilisation de wrappers, il est bien sûr nécessaire d'effectuer une réécriture du code. Cela peut donc être d'autant plus coûteux.

Si une solution telle qu'OpenACC semble très prometteuse en vertu de sa capacité à cibler différentes architectures, tout en étant le moins intrusif possible, en pratique, le fait de ne pas pouvoir prendre un code C++ en entrée demande dans tous les cas une phase de réécriture du code pour le préparer à cette API. Une question se pose donc quant à l'utilisation d'une API telle qu'OpenCL qui nécessite dans tous les cas une réécriture des codes mais permet une approche plus bas-niveau et donc potentiellement plus d'optimisations.

1.4.2 Envrionnement multithread

La plupart des logiciels ont été développés pour des plateformes monocoeur. L'utilisation de threads était rare et n'était que peu utilisée dans un but de gain de temps, mais plutôt en terme d'organisation de code (ex : modèle producteur-consommateur ou MVC pour les interfaces graphiques). Les applications aujourd'hui sont donc généralement un mélange de code séquentiel statique et d'un modèle objet. En JAVA par exemple, la partie affichage via Swing, utilise un thread principal permettant à l'API d'être *thread-safe*, c'est-à-dire qu'une fonction puisse être appelée depuis plusieurs threads parallèlement, même si cette fonction utilise des données partagées, car ces threads vont être exécutés de manière séquentielle.

Lors de l'intégration d'un code OpenCL par exemple, il est nécessaire de prendre en compte le fait que la fonction `clSetKernelArg()` n'est pas *thread-safe*. C'est la seule fonction de l'API qui est dans ce cas, lorsque la même variable *kernel* est utilisée. Ce type d'appels est donc à gérer, de manière séquentielle, ou alors à éviter lorsque cela est possible.

Ces questions existent bien évidemment déjà pour des codes non optimisés, mais l'utilisation de nouvelles bibliothèques est à prendre en compte et ajoute de la complexité à l'application, et donc d'éventuels bugs.

1.4.3 Coût de développement

Les technologies utilisées pour l'optimisation, telles que le SIMD ne sont pas très jeunes : les machines vectorielles remontent aux années 70 et le calcul SIMD, apparue sur ces dernières, n'a réellement été intégré les processeurs grand public qu'en 1996 avec MMX. En revanche, la parallélisation n'est arrivée dans le grand public qu'avec l'arrivée des multicoeurs d'AMD et d'Intel en 2006. Les technologies utilisées à ce niveau sont donc forcément relativement jeunes. Le compilateur GCC par exemple, sorti en 1987, est un outil qui a énormément évolué depuis. Lorsque l'on compare ce compilateur à celui de Nvidia pour CUDA, il semble logique qu'il ne puisse pas être au même niveau que celui de GCC, d'Intel ou même de celui de Microsoft. Nvidia a d'ailleurs fait un pas important avec CUDA 4.1 en intégrant le compilateur LLVM, améliorant significativement les performances de ses applications par la même occasion.

Ainsi, les outils disponibles au niveau parallélisme sont évidemment moins aboutis que ceux existants pour l'optimisation de code séquentiel. Il est donc moins aisé de développer des applications parallèles. Cela peut paraitre évident, mais cette assertion engendre des coûts de développement plus importants et qui sont à prendre en compte dans le rapport coût/performance. Ce coût est d'ailleurs d'autant plus compliqué à évaluer, du fait du manque d'outils de débogage en OpenCL. Cette API dispose d'une gestion des erreurs pour la plupart de ses primitives, mais le problème est que les implémentations réalisées, par Nvidia, AMD, ou Intel, ne renvoient pas toujours un code d'erreur compréhensible, ou pire, renvoient une erreur à la primitive suivante car la précédente n'a pas été détectée comme telle.

Ces problèmes peuvent faire perdre énormément de temps au développeur et nécessitent une grande rigueur et de la patience pour les développeurs. Il n'est donc pas toujours évident pour un industriel de faire un choix allant vers des technologies aussi jeunes.

1.4.4 Maintenance, évolutions, pérennité

Dans un registre très lié au coût, la maintenance est elle aussi au centre des problématiques de développement d'un logiciel industriel. En effet, les développements s'étalent parfois sur plusieurs décennies. Par conséquent, avant d'intégrer de nouvelles technologies, on cherche donc à évaluer son utilisabilité à moyen terme. Nombreuses sont les questions qui se posent : l'API utilisée permet-elle de cibler plusieurs architectures ? Et sera-t-elle fonctionnelle sur les architectures matérielles à venir ? Les développeurs seront-ils montés en compétence sur le sujet ? etc...

Les réponses à ces questions ne sont pas aisées. Si l'évolution des architectures s'oriente vers le parallélisme massif, rien ne nous permet vraiment de savoir si les GPU seront encore

utilisés pour du GPGPU d'ici 5 à 10 ans, d'autant plus que leur fonctionnement reste relativement proche de celui d'une boite noire, là où les GPP ont pu être décortiqués depuis de nombreuses années. En revanche, les GPP ont un modèle de développement qui correspond à leur utilisation : ils évoluent le plus possible de manière à minimiser les impacts négatifs sur les codes. A part le tournant pris sur le parallélisme, les GPP ont toujours cherché à améliorer leurs performances de manière globale. A part la multiplication des coeurs, Intel et AMD augmentent la taille de leurs caches, améliorent le pipeline d'exécution et augmentent la taille des registres SIMD. Ces modes d'évolution sont donc relativement pérennes.

Les GPU, Nvidia ou AMD, cherchent eux aussi à évoluer dans ce sens. L'architecture Nvidia, depuis 2006, est modifiée à bas niveau mais reste identique à haut niveau, de manière à gêner le développeur le moins possible.

Des outils tel qu'OpenMP, demandant des modifications de la structure du code minimales semblent être une contrepartie intéressante. OpenMP va permettre de rapidement profiter du parallélisme d'une architecture tout en demandant un minimum de développements. De plus les compilateurs sont de plus en plus à même d'effectuer de la vectorisation. Cette solution alliant OpenMP à la vectorisation semble être une solution des moins coûteuses et des plus pérennes étant donné l'évolution du marché des GPP actuellement. L'un des facteurs principaux est donc bien l'évolution des architectures matérielles qu'il est nécessaire de suivre de près.

1.4.5 Matériel

Les architectures matérielles évoluent rapidement entrainant des modifications dans les API permettant de les programmer, voire même l'arrivée de nouvelles API. L'évolution du matériel est à la source des problèmes de maintenance et de pérennité.

Prenons l'exemple des GPU : les transferts vers ces accélérateurs sont un vrai goulot d'étranglement pour un grand nombre d'applications. Bien que la norme du port PCI-Express, lequel est utilisé par les GPU, ait énormément évolué depuis 2006, passant d'une bande passante de 4 Go/s à 16 Go/s dans chaque sens, le fait que le GPU ne puisse pas utiliser le même espace mémoire est problématique. Nvidia propose déjà d'utiliser l'espace mémoire du GPP au travers d'une technologie telle que *ZeroCopy*, mais les latences d'accès sont très handicapantes.

AMD a donc mis sur le marché en 2011, un APU (Accelerated Processing Unit), agrégation d'un GPP et d'un GPU sur la même puce. L'objectif principal est de rapprocher le GPU, capable d'exécuter des codes très réguliers et intensifs de manière très efficace, afin de minimiser les transferts et décharger le GPP de certains calculs.

Actuellement, l'espace mémoire n'est pas partagé. Le GPP et le GPU disposent d'un espace virtuel chacun, et il est donc nécessaire d'effectuer des transferts manuellement. En revanche, AMD devrait apporter un adressage homogène commun en 2013 au travers d'un contrôleur mémoire unique et devrait aussi apporter plus de souplesse avec une gestion des changements de contextes par exemple. Quelles que soient les évolutions, l'idée de rapprochement entre GPP et GPU est intéressante et demande à être suivie. Permettre au GPP d'effectuer des tâches précises de manière plus efficace est prometteuse. En revanche, il est encore tôt pour savoir si de manière logicielle, ces solutions pourront être adoptées par une large communauté de développeurs. Cela passera obligatoirement par la création de bibliothèques de fonctions profitant

d'une accélération matérielle de la part des coeurs GPU.

Parmi les contraintes matérielles, il existe aussi des problèmes plus anecdotiques, comme par exemple pour les GPU. Ces cartes sont volumineuses et consomment plus de 200W pour le haut de gamme. Si la majorité des utilisateurs peut se contenter d'une carte entrée de gamme, le GPGPU va demander la plupart du temps de disposer de cartes haut-de-gamme, sous peine de ne pas pouvoir réellement profiter de bonnes performances. Il n'est pas toujours aisé de placer une carte GPU consommant parfois plus de 300W dans une station de travail créée pour un GPU de 40W. Il est bien souvent nécessaire de changer de matériel au complet pour pouvoir utiliser ce type de cartes. Lorsque ce changement va se poser pour un parc entier de machines, le coût peut devenir très vite limitant. L'argument avancé par Nvidia, qui était de dire que tous les ordinateurs de bureaux sont équipés de GPU inutilisés, est uniquement marketing.

1.4.6 Perspectives par rapport à la plateforme CIVA

Dans le cadre de la perspective d'intégration dans la plateforme CIVA, plusieurs tendances ressortent. Les langages de programmation utilisés pour le développement sont principalement JAVA et C++. Le logiciel existe depuis plus de 10 ans et atteint les deux millions de lignes de code.

Dans le cadre de l'accélération des codes de calculs, il est donc nécessaire d'évaluer différentes possibilités. Avec comme cible les ordinateurs de bureau, les cibles sont relativement restreintes et sont principalement les GPP multicoeurs et les GPU. Ce sont les deux solutions les plus abouties actuellement en vue de l'intégration de codes parallélisés dans un contexte industriel tel que celui de CIVA. Evidemment, le recul sur les GPU est très limité, et si les GPP multicoeurs ne sont pas près de disparaitre, il est pertinent d'explorer les pistes existantes, telle que celle du GPU.

Concernant les langages, les GPU sont utilisables au travers de différentes API, de l'API native, telle que CUDA, au langage basé sur des *pragmas*. Le C++ n'était pas directement pris en charge, le choix d'OpenCL parait intéressant, dans la mesure où il a été créé pour piloter différentes architectures, et donc permettant d'obtenir une pérennité des codes plus importante.

1.4.7 Mesures de performances

Afin de mesurer les accélérations, il est nécessaire d'avoir des moyens de mesures de performances fiables et stables.

Toutes les architectures et outils proposent différentes méthodes de mesures de performances. Sur GPP, un temps peut être mesuré par différents moyens. Lorsque l'on cherche à mesurer des temps qui peuvent descendre sous la milliseconde, il est préférable d'utiliser un *timer* haute-résolution. Sous Windows, le *timer* QueryPerformanceCounter (QPC) dispose d'une résolution de l'ordre de la microseconde. Ce *timer* est donc utilisable pour un large panel de mesures contrairement à une fonction telle que `timeGetTime` qui ne dispose que d'une résolution de l'ordre la dizaine de millisecondes.

Sur une plateforme Linux, la norme POSIX propose deux timers haute-résolution. Le premier dispose d'une résolution à la microseconde : `getTimeofDay`, et le second se base sur

`gethrtime` et dispose d'une résolution à la nanoseconde.

Il existe aussi des *timers* matériels. Sur GPU par exemple, les *events* de CUDA utilisent un registre mis à jour de manière matérielle et qui permet d'obtenir une résolution élevée (0,5 microsecondes). Quant au GPP, il est possible d'utiliser RDTSC pour les GPP Intel.

Les mesures réalisées dans ce manuscrit utilisent QPC pour les GPP et les *timers* matériels des GPU. Les temps utilisés dans les différents tableaux et graphiques sont les temps minimum sur une séquence de 10 exécutions du programme.

1.4.8 Conclusion sur les problématiques d'intégration

Les problématiques d'intégration de codes optimisés et parallélisés sont nombreuses. De plus, ces problématiques ne peuvent pas aisément être adressées. L'évolution des langages et des outils, ainsi que l'installation sur la durée de ces derniers est nécessaire pour obtenir une montée en compétence des développeurs, nécessaire à la pérennité du code.

Dans cet écosystème mouvant, OpenCL et OpenACC semblent être les solutions à suivre. D'un côté, OpenCL propose une approche bas-niveau, tandis qu'OpenACC passe par des directives telles qu'OpenMP et met principalement en avant le découpage et le parallélisme de l'application en restant haut-niveau.

S'il est difficile pour un industriel de lancer des développements à grande échelles sur ces nouvelles technologies, il est tout à fait possible d'intégrer ce type d'outils à un niveau de l'application plus bas, permettant de minimiser les modifications d'interface dans le code et d'interface utilisateur. C'est cette approche que nous avons choisie pour effectuer l'intégration dans la plateforme CIVA des algorithmes étudiés et optimisés dans les chapitre 2 et 3 du manuscrit.

1.5 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté le contexte métier de la thèse. Nous avons pu observer sa richesse et sa complexité et nous avons expliqué les motivations du travail décrit dans ce manuscrit.

Ensuite, nous avons présenté le contexte en termes d'architectures, principalement des deux les plus adaptées à l'accélération de codes industrialisés : les GPP et les GPU. Les GPP sont installés sur le marché depuis le début de l'informatique, mais le tournant technologique du parallélisme pris en 2006 a remis en cause l'évolution des performances apportée ici principalement par l'augmentation de fréquence. Il est donc nécessaire aujourd'hui d'exploiter le parallélisme sur ces architectures. Les GPU existent depuis déjà longtemps, mais c'est aussi depuis 2006 qu'il est possible de les programmer au travers d'API dédiées. Le recul obtenu par la communauté scientifique ainsi que l'exploitation de ces architectures dans des logiciels industriels permet de penser qu'il est raisonnable d'utiliser ces architectures.

Cependant, un ensemble de problématiques se posent quant à l'utilisation de ces nouvelles architectures. Tout d'abord, les outils de programmation arrivent certes à maturité, mais la

grande quantité de langages utilisés fait qu'il est souvent nécessaire de passer par des API faisant le pont entre le langage utilisé par l'application et le langage natif utilisé sur l'accélérateur. Ces API ne sont généralement pas officielles et leur maintien n'est pas forcément assuré. Ensuite, en terme de coûts de développement, il existe aujourd'hui une marge importante entre le développement d'un code séquentiel et celui d'un code parallélisé. Sur GPP, des outils tels qu'OpenMP permettent d'obtenir des performances dans des temps de développement relativement rapides. En revanche, dès que l'on attaque du GPU, il est nécessaire d'effectuer une phase d'optimisation du code importante pour obtenir une utilisation efficace de l'architecture. Avec un coût encore plus important, se trouve l'utilisation des instructions SIMD des GPP, qui nécessitent une connaissance encore plus fine de l'architecture pour obtenir de bonnes performances, pourvu que le code soit suffisamment régulier pour être performant. Ces coûts sont donc à prendre en compte.

Le tournant du parallélisme ayant été pris, il semble peu probable de voir un retour à du code séquentiel à court voir même moyen terme. En revanche, l'avenir d'architectures déportées du GPP, telles que les GPU ou le Xeon Phi est sûrement moins clair. En effet, les transferts nécessaires pour utiliser ces architectures sont coûteux et les vitesses de bus n'augmentent pas assez vite par rapport à la puissance de calcul. Un rapprochement des accélérateurs, pour obtenir une enveloppe unique intégrant une partie GPP et GPU, telle que propose AMD avec Fusion, semble donc être une voie prometteuse. Le milieu du parallélisme bouge donc très vite, et s'il est tout à fait pertinent d'effectuer des recherches sur la parallélisation de code aujourd'hui, il n'est pas évident de dire quelle architecture doit être utilisée, dans la mesure où la pérennité n'est pas assurée.

CHAPITRE 2

ETUDE DE LA PARALLÉLISATION DE L'ALGORITHME IMAGE VRAIE CUMULÉE

Ce chapitre traite de la première étude réalisée dans le cadre de la thèse. Nous avons vu qu'il existe une famille d'algorithmes de reconstruction qui utilisent une approximation de type "rayon" pour la simulation de la direction de propagation des ondes dans la pièce. Nous allons étudier l'algorithme "Images Vraies Cumulées" qui a la particularité de traiter l'ensemble des indications issues d'une acquisition ou d'une simulation. La nécessité de traiter un grand nombre de données engendre des temps de calculs importants et limite considérablement l'utilisation de cet algorithme dans un contexte industriel.

Les objectifs sont donc de proposer une alternative algorithmique aux résultats équivalents, mais avec des performances sensiblement meilleures en exploitant au mieux les architectures parallèles. Une alternative à l'algorithme initial proposé par le DISC sera présentée et parallélisée, les différentes approches seront étudiées et comparées, enfin on évaluera les contraintes et possibilités d'intégration dans une version commerciale du logiciel CIVA. Les travaux réalisés dans ce chapitre ont fait l'objet de deux publications dans les conférences DASIP 2011 et PARCO 2011 [Pédron et al., 2012] [Pédron et al., 2011].

2.1 Présentation de l'algorithme d'Images Vraies Cumulées

Dans cette partie, nous commencerons par présenter l'algorithme Images Vraies Cumulées, puis nous détaillerons l'implémentation existante et les formats de données utilisés. Nous terminerons par une mise en évidence des approches de parallélisation.

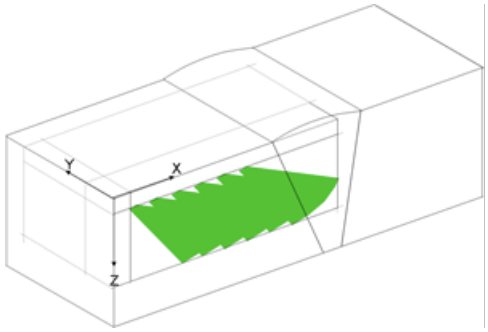
2.1.1 Présentation fonctionnelle

L'algorithme d'Images Vraies Cumulées (IVC) est un algorithme de reconstruction ultrasons basé sur la notion de direction de propagation (que nous appellerons aussi trajet - c.f.

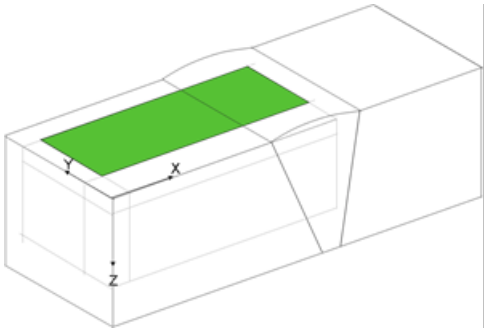
section1.1.3.4). Cet algorithme a pour but de localiser les échos dans une pièce et de les dimensionner afin de caractériser les défauts. Il permet d'obtenir une vision complète de l'ensemble du contrôle via les vues de côté, de face et du dessus de la boite englobant la zone contrôlée de la pièce.

Cet algorithme repose sur le fait que la géométrie de la pièce, ainsi que les propriétés du matériau sont connues. La géométrie de la pièce est donnée soit de manière analytique pour les cas simples (parallélépipède rectangle, cylindre), soit par une description CAO (profil 2D extrudé par translation ou rotation).

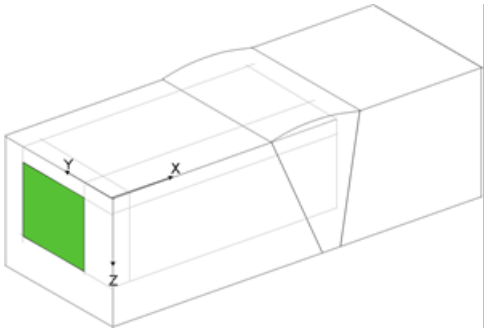
Si l'on considère un *B-Scan* vrai tel qu'il a été présenté dans le chapitre 1, le *B-Scan* vrai cumulé obtenu par l'algorithme d'IVC correspond à la projection, par un calcul de maximum d'amplitude, de l'ensemble des signaux du contrôle repositionnés, sur la face de côté de la boite-englobante du contrôle. Le même principe s'applique pour les C-Scan et les D-Scan. La figure 2.1 présente ces trois vues par rapport à une pièce plane. Les vues peuvent ensuite être replacées dans la vue 3D afin de bénéficier des outils de visualisation de CIVA.



(a) Vue de côté



(b) Vue de dessus



(c) Vue de face

FIGURE 2.1 – Positionnement des vues obtenues par l'algorithme IVC dans la vue 3D. On distingue la boite-englobante du contrôle en traits gris clairs sur les différentes vues.

Le calcul consiste ici à replacer l'ensemble des signaux acquis sur leur trajet correspondant. La figure 2.2 montre ce principe en 4 étapes.

Afin d'obtenir une cartographie de la pièce que l'on souhaite contrôler, le contrôle intègre un déplacement mécanique du capteur, et/ou un balayage électronique des voies du capteur.

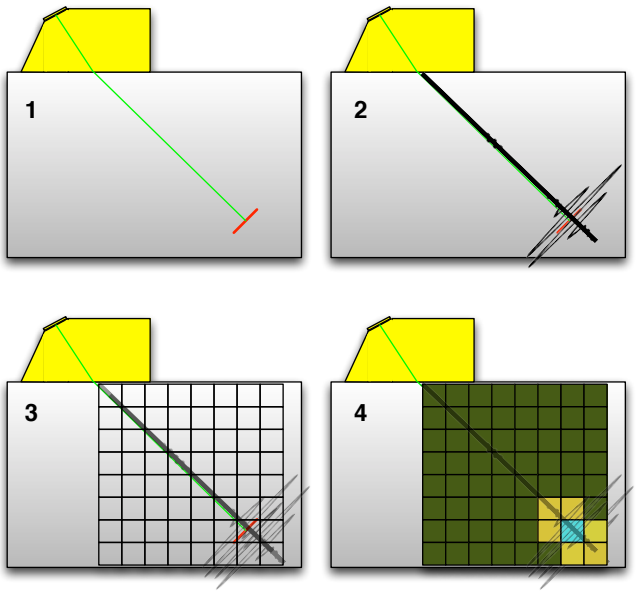


FIGURE 2.2 – Schématisation des étapes principales de l'algorithme d'Images Vraies Cumulées. L'étape 1 montre un rayon approximant la propagation de l'onde ultrasonore dans la pièce en vert et un défaut en rouge. A l'étape 2, le signal obtenu en acquisition est transformé dans le domaine cartésien du trajet. A l'étape 3, le trajet est projeté dans l'espace 2D de visualisation. Enfin, à l'étape 4, l'information d'amplitude est cumulée par un calcul de maximum (représenté ici avec une échelle de couleur inspirée de celle de CIVA).

Pour chaque position mécanique, on acquiert un signal supplémentaire. L'algorithme IVC va accumuler l'ensemble des signaux et les cumuler par un calcul de maximum, comme nous venons de le voir ci-dessus.

Ce type d'accumulation va avoir un impact sur la visualisation de l'image. En pratique, l'espacement entre deux positions d'un balayage mécanique peut varier entre chaque contrôle et est indépendant du maillage pixel de l'image. De plus, l'information d'amplitude qui est projetée sur l'image n'est initialement que ponctuelle, tandis que d'un point de vue physique, l'information n'est pas réellement ponctuelle. Un remplissage des zones "vides" est donc nécessaire. Les algorithmes et méthodes de remplissage seront étudiées par la suite.

2.1.2 Pseudo-code de l'algorithme et implémentation de référence

En entrée de l'algorithme, on trouve les données suivantes :

- N_t signaux temporels de taille fixe n_E. Ces signaux sont issus, soit d'une acquisition, soit du calcul de simulation de la plateforme CIVA.
- N_t trajets : ces trajets sont une approximation géométrique de la propagation de l'onde ultrasonore dans le milieu contrôlé. Un trajet se décompose en une suite de segments dont le nombre varie selon les interfaces rencontrées. Ces trajets sont issus du module de simulation de la plateforme CIVA.
- Une zone 2D correspondant à la zone de reconstruction dans le repère de la pièce.

Algorithme 1: Pseudo-code de l'implémentation de référence de l'algorithme IVC

entrée : tableau de trajets $Traj$
entrée : tableau de signaux $Sigs$
entrée : nombre de signaux N_t
entrée : pas d'échantillonnage des signaux Δt
entrée : taille de la zone de remplissage $(\Delta u, \Delta v)$
sortie : image bitmap Im de taille $n \times m$

```
// Boucle sur les trajets
1  for i ← 1 to n_E do
2      nS ← nombre de segments de Traj(i)
       // Boucle sur les segments du trajet
3      for j ← 1 to nS do
4          M_j^i, M_{j+1}^i ← calcul des points extrémités de Traj(i, j)
5          t_j^i, t_{j+1}^i ← calcul des temps de vol aux points M_j^i, M_{j+1}^i
6          nS ← (t_{j+1}^i − t_j^i)/Δt
           // Boucle sur les échantillons du signal correspondant au segment
7          for k ← e_k of seg(i, j) to nS do
8              a ← extraction de l'amplitude Sigs(i, k)
9              P(x, y, z) ← calcul de la position de l'échantillon k sur seg(i, j)
10             P'(x', y') ← projection 3D→2D de P
               // Boucle sur les pixels de la zone à remplire (: réduction)
11             for v ← −Δv to Δv do
12                 for u ← −Δu to Δu do
13                     Im[y' + v, x' + u] ← max(a, Im[y' + v, x' + u])
```

En sortie, on reconstruit une image où la valeur de chaque pixel correspond à une amplitude.

2.1.3 Description des données

Les données issues d'une simulation ou d'une acquisition sont stockées dans des fichiers dont le format dépend des systèmes d'acquisition. Lors d'une reconstruction classique, les signaux sont chargés à la volée depuis ces fichiers, à travers des API et bibliothèques généralement fournies par le constructeur du système d'acquisition. Dans la mesure où l'on ne peut pas maitriser les performances d'accès aux données sans les convertir dans un format propre, et pour s'affranchir des problématiques d'accès disque, il a été décidé de charger l'ensemble des données en DRAM. Compte tenu des capacités mémoire des architectures 64 bits, et des ordres de grandeurs des fichiers d'acquisition CND, ce choix est compatible avec la plupart des contextes d'utilisation de la plateforme logicielle CIVA.

2.1.3.1 Signaux

Afin d'optimiser l'accès mémoire, les signaux sont rangés de manière contigüe et sont tous redimensionnés sur une même fenêtre temporelle. Il est à noter que la fréquence d'échantillonnage est homogène et constante pour une acquisition donnée. Pour réduire l'espace nécessaire au stockage, les amplitudes nulles peuvent être compressées, mais nous ne prendrons pas en

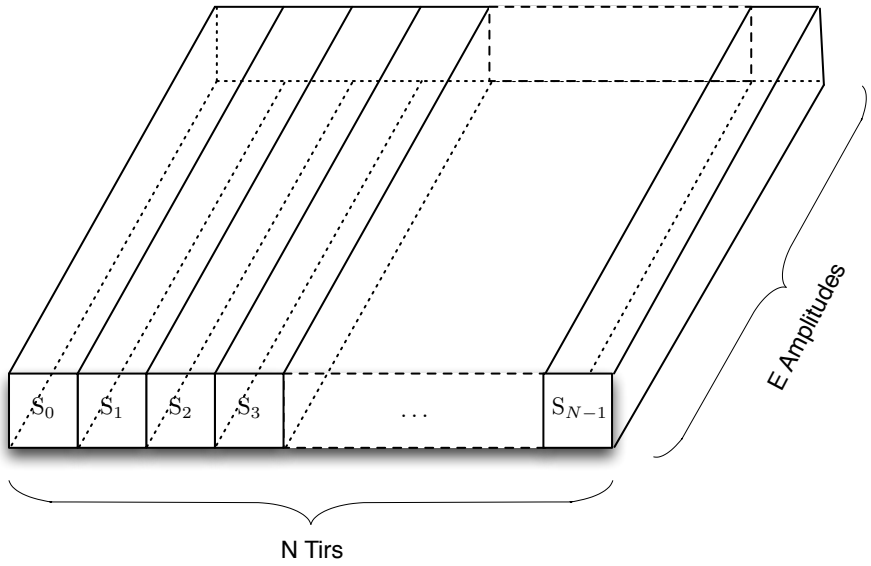


FIGURE 2.3 – Matrice des signaux stockée de manière linéaire en mémoire.

compte ce type d'optimisation. On a donc une structure mémoire régulière telle que celle présentée sur la figure 2.3 pour N_t tirs ultrasons et donc N_t signaux de N_e amplitudes.

Les signaux sont une suite d'échantillons numériques stockés soit en flottants 64 bits (données issues de la simulation), soit en entiers sur 16 bits (données issues des acquisitions).

La taille des signaux est variable entre deux contrôles et peut aller de quelques centaines d'échantillons à une dizaine de milliers. Cela peut donc représenter dans certains cas, plusieurs Go de données, puisque l'on peut envisager plusieurs dizaines de milliers de tirs lors d'un contrôle.

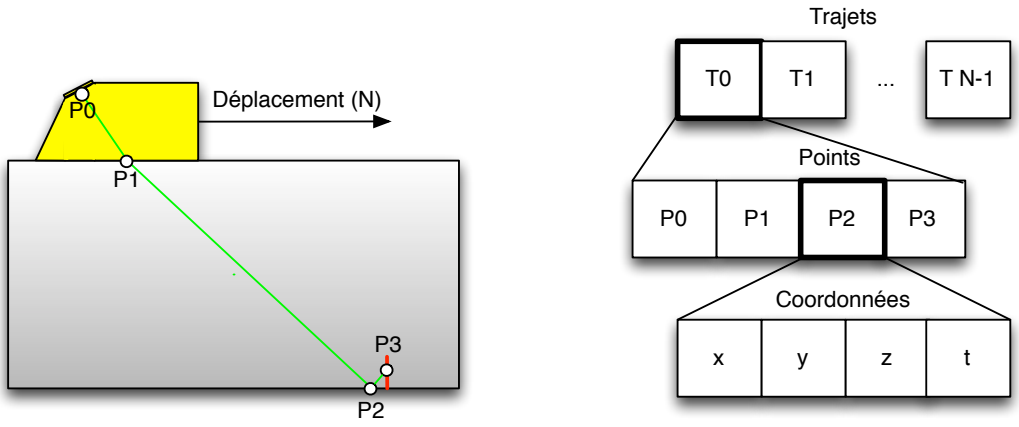
2.1.3.2 Trajets

Les trajets sont issus du module de calcul de trajets de la plateforme CIVA. Pour notre étude nous les considérons comme une entrée de notre problème. En effet, comme nous l'avons dit dans la section 1.1.3.4, pour un contrôle donné, les trajets sont précalculés par CIVA via un module de lancer de rayons et sont stockés dans un fichier.

Les trajets sont une polyligne qui représente une approximation du chemin de propagation de l'onde ultrasonore dans la pièce. Pour chaque tir, un rayon est lancé vers la pièce et est propagé selon la loi de Snell-Descartes. La modélisation par trajet émet l'hypothèse que l'onde suit le même parcours à l'émission et à la réception.

On peut voir, sur partie (a) de la figure 2.4, le parcours d'un trajet dans une pièce simple où l'on a placé un défaut. Le trajet se décompose ici en 3 segments. Ce nombre est variable et dépend de la géométrie de la pièce. Par exemple, dans le cas d'une soudure, on modélise les différents volumes de la soudure comme ayant une interface. On a donc des variation de trajets qui peuvent être importantes dans ces zones.

Toujours sur la figure 2.4, sur la partie (b), on peut voir la manière dont les trajets sont or-



(a) (b)

FIGURE 2.4 – a) Parcours d'un trajet dans une pièce. b) Ordonnancement des trajets en mémoire. Les trajets sont ordonnés par tirs, puis par points représentant les extrêmités de chaque segments. Chaque point est représenté par sa position dans le volume de la pièce *(x, y, z)* et un temps de vol à l'interface t. Le nombre de points par trajet est variable et dépend de la géométrie de la pièce et de la position du capteur.

donnancés en mémoire. Chaque trajet est représenté par une liste de points correspondant aux extrémités des segments le composant. On associe à cette information géométrique le temps vol : temps nécessaire à l'onde pour atteindre ce point compte tenu la vitesse de propagation des ondes dans les matériaux et de la distance parcourue le long du trajet. C'est grâce à cette information temporelle associée au trajet géométrique que l' on va être capable ensuite de replacer les amplitudes des signaux dans l'espace cartésien de la pièce.

2.1.3.3 Zone de reconstruction

Les autres paramètres sont ceux liés à la géométrie et à l'image. En effet, comme nous l'avons noté, il est nécessaire de connaitre la zone de reconstruction dans la pièce, donc dans le repère physique, ainsi que la résolution de la grille image sur laquelle on va travailler.

2.1.4 Complexité algorithmique

L'algorithme doit accéder à l'ensemble des signaux et trajets associés. Pour un contrôle de N_t couples trajet/signal avec N_e échantillons par signal, on a une complexité algorithmique en $O(N_t \times N_e)$ en terme d'accès mémoire.

En terme de calcul, le tableau 2.1 montre l'ordre de grandeur des lecture/écriture et des calculs pour les différentes boucles, ainsi que l'ordre de grandeur du nombre d'itérations pour des cas d'utilisation réels. La boucle trajet est la boucle que l'on retrouve sur l'algorithme 1 à la ligne 1, la boucle segment se trouve à ligne 3 et enfin, la boucle échantillon se retrouve à la ligne 7. On constate que la boucle interne est celle qui nécessite le plus de calcul avec 32 opération arithmétiques contre 2 à 3 accès en mémoire. On a donc un algorithme dont le nombre de calculs est relativement important face aux entrées/sorties avec un ratio d'environ 3 I/O pour 32 opérations arithmétiques soit une intensité arithmétique supérieure à 10. Ainsi, nous devrions obtenir des implémentations limitées par la puissance de calcul et non par la vitesse des accès

à la mémoire.

	Load	Store	Arith.	Intervalle terations
Boucle Trajets (algo. 1 ligne 1)	2	0	1	[10k - 200k]
Boucle segments (algo. 1 ligne 3)	8		12	[2-10](*)
Boucle echantillons (algo. 1 ligne 7)	2	1(**)	32	[30-10k]

Tableau 2.1 – Quantités de lectures/écritures, nombre d'opérations arithmétiques et intervalles de dimensionnements pour les trois boucles de l'algorithme IVC.
(*) Valeur variable pour un contrôle ultrasonore donné (**) le calcul d'un maximum nécessite une lecture, une comparaison et éventuellement une écriture.

L'autre point important que l'on peut constater sur le tableau 2.1 est la disparité dans le nombre d'itérations des boucles. Pour un même contrôle et donc une même reconstruction, ces variations vont entrainer des irrégularités dans les flux de calculs. En effet, selon le contrôle effectué, on peut avoir des cas avec un très grand nombre de trajets, deux segments par trajet, et un grand nombre d'échantillons, comme aussi un nombre variable de segments par trajets, et peu d'échantillons. Même si cela ne change pas radicalement la complexité de l'algorithme, cela implique des variations sur la boucle interne qui peuvent être importantes.

2.2 Optimisation du traitement d'image

Dans cette partie, nous allons nous intéresser au traitement d'images et à la manière dont il est possible de l'optimiser. En effet, comme nous allons le voir, une phase d'analyse nous a permis de noter le coût très important engendré par l'algorithme initial. Nous commencerons par décrire le traitement existant puis nous proposerons une solution d'accélération de ce calcul.

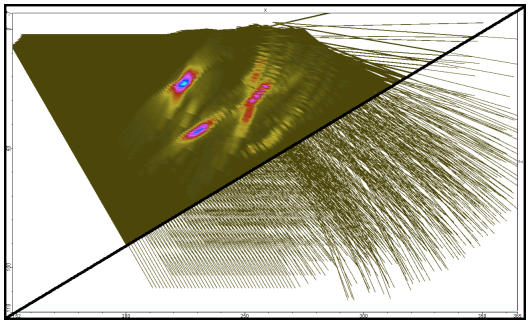
2.2.1 Description du traitement d'image existant

Dans la section 2.1, nous avons décrit le traitement d'image qui est appliqué lors de la projection. Nous allons maintenant détailler le processus pour les différents cas métier.

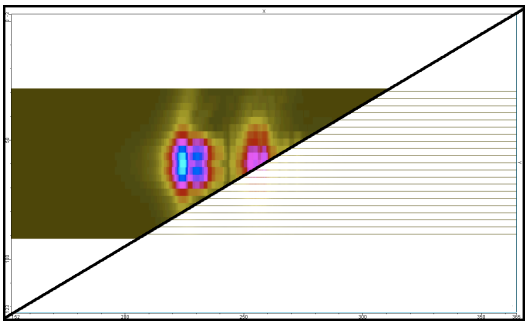
On dimensionne chaque point projeté par une zone rectangulaire en millimètres et on applique l'algorithme d'accumulation de maximum en cas de recouvrement. En pratique les dimensions sont ajustables par l'utilisateur de l'algorithme, mais il convient d'utiliser des dimensionnements correspondants aux valeurs suivantes :

- Pas de balayage/incrément/déplacement mécanique
- Pas de balayage/incrément/déplacement électronique
- Résolution spatiale du signal

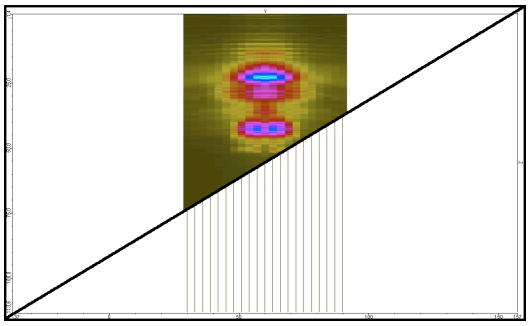
Ce traitement permet d'obtenir des images remplies telles que l'on peut voir sur la figure 2.5. Ces images sont issues ici de reconstructions provenant d'une simulation sur un modèle de pièce de type soudure.



(a) Vue de côté



(b) Vue de dessus



(c) Vue de face

FIGURE 2.5 – Reconstructions issues de l'algorithme IVC présentant les trois vues avec et sans traitement. On constate bien l'aspect ponctuel des amplitudes lorsqu'il n'y a pas de traitement.

2.2.2 Remplacement du traitement par un opérateur de morphologie mathématique

L'approche de remplissage présentée précédemment remplit de manière satisfaisante l'image pour l'utilisateur en propageant l'information d'amplitude sur une zone où il n'a initialement pas d'information. Cependant, un calcul de maximum s'avère relativement coûteux et le recouvrement entre deux amplitudes propagées peut engendrer un nombre de calcul de maximums très importants. Une phase de *profiling* du code nous a permis de mettre en évidence ce coût, que nous préciserons dans la suite de cette section.

Pour cette raison, nous nous sommes demandés s'il était possible d'appliquer un traitement équivalent en post-traitement tout en réduisant le coût de recouvrement. On a donc regardé parmi les traitements existants lequel pourraient nous permettre d'obtenir un résultat équivalent voire identique. Nous avons cherché quel type d'algorithme permettait de "boucher des trous".

Nous avons tout d'abord regardé du côté des algorithmes d'*inpainting* mais ceux-ci sont généralement faits pour restaurer de l'information perdue ou corrompue dans de petites zones de l'image [Paragios et al., 2005]. Ce type de filtre n'est donc pas propice étant donné que les zones vides peuvent être de taille importante. Nous avons ensuite regardé du côté des opérateurs locaux.

Nous avons constaté que l'opérateur de morphologie mathématique de dilatation était exactement ce que ce que nous recherchions. Cet opérateur a été initialement développé pour

des images binaires et a depuis été étendu pour des images en niveaux de gris ou en couleur. L'opération se résume de la manière suivante :

Le noyau de l'opérateur est déplacé sur chaque pixels de l'image "trouée". A chaque pixel (x, y), l'opérateur détermine la valeur maximum sous la zone recouverte par le noyau et l'écrit dans le pixel de l'image en cours de remplissage.

On obtient un étalement de l'information de la même manière que le traitement à la volée initial, mais cette fois-ci en post-traitement. On se base sur l'hypothèse que le traitement est de taille fixe pour tous les pixels ainsi que de l'associativité, de la commutativité et de l'idempotence de l'opérateur de maximum. On passe donc d'un algorithme de remplissage de complexité en $O(N_t \times N_e)$ à un algorithme en $O(N_p)$. Il se trouve que dans les cas qui nous intéressent, le nombre de pixels de l'image est largement inférieur au nombre d'échantillons du contrôle ce qui devrait permettre au nouveau traitement d'être d'autant plus performant.

Il est à noter que le post-traitement ne peut donner rigoureusement le même résultat que le traitement à la volée. En effet, lors de l'application à la volée, le centre du traitement est placé non pas sur un pixel, mais sur une position géométrique 2D. Cela implique que le noyau peut déborder sur un pixel sur lequel il ne déborderait pas lors du post-traitement, puisqu'on se place au centre du pixel dans ce cas. La figure 2.6 présente ce cas. L'erreur étant de l'ordre de l'arrondi sur un pixel, on la considère comme non significative.

Ce type de traitement est connu et il est possible d'utiliser des optimisations classiques du traitement d'images, telles que la séparabilité du noyau. En effet, comme pour un noyau Gaussien rectangulaire, il est possible de séparer le calcul en deux passes (pour un filtre 2D) : une passe horizontale et une seconde verticale. Cela permet de réduire le nombre de calculs [Saidani et al., 2011]. De plus, cette étape peut être optimisée de manière encore plus poussée comme il est décrit dans [van Herk, 1992] [Domanski et al., 2009]. Ce type d'optimisations est par exemple utilisé dans d'autres types d'algorithmes de projection tel que l'algorithme de projection d'intensité maximale [Roerdink, 2003]. Etant donné le gain de performances apporté par la modification du traitement, il serait possible de donner une forme circulaire au noyau afin d'obtenir un traitement plus proche de la réalité physique. Ce traitement n'a pas été fait car comme nous le verrons dans la partie intégration, un autre type de traitement a été adopté.

2.3 Présentation du benchmark

Dans cette section, nous allons détailler en première partie le matériel et les outils utilisés pour les benchmarks. En deuxième partie, nous présenterons les jeux de données que nous avons choisis pour effectuer ces benchmarks.

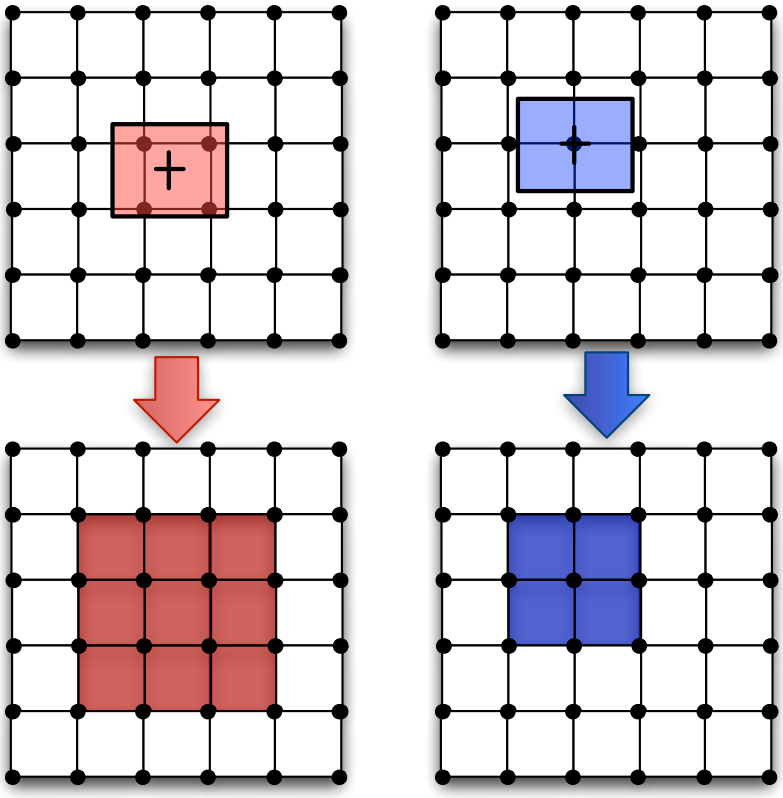


FIGURE 2.6 – Différence entre le traitement à la volée (en rouge) et le post-traitement (en bleu). Ici est présenté un cas particulier où l'alignement de la zone pour le post-traitement est de dimension inférieure à celle pour le traitement à la volée.

2.3.1 Aspects logiciels et matériels

Etant donné l'évolution rapide de certains pilotes ou des API (API CUDA : 4 versions majeures et 7 mineures en 5 ans), il est parfois problématique de reproduire à l'identique certains benchmarks. Pour cette raison, nous souhaitons présenter les versions de pilotes ainsi que d'API utilisés pour les benchmarks.

Concernant les GPU, la version de l'API utilisée a été CUDA 4.0 avec le driver 270.51. Trois GPU Nvidia de deux générations différentes ont été utilisés pour réaliser les benchmarks. Du côté des GPP testés, nous en avons aussi évalué deux de deux générations différentes. Les spécifications principales de ces architectures sont visibles dans le tableau 2.2.

Tableau 2.2 – Spécifications principales des architectures évaluées pour l'algorithme IVC.

Modèle	# coeurs	Fréq. (GHz)	BP Mémoire (GB/s)	GFLOPs SP (scalar/SIMD)	GFLOPs DP (scalar/SIMD)
Intel E5472 - Penryn	4	3,00	12,8	24/96	24/48
Intel X5690 - Westmere-EP (2)	2x6 (2x12 threads)	3,47	64	83/332	83/166
NVIDIA GeForce GTX 580	512 (16 SM x 16 c.)	1,54	192	1580	198
NVIDIA Tesla C2070	448 (15 SM x 16 c.)	1,15	143	1030	515
NVIDIA Tesla C1060	240 (30 SM x 8 c.)	1,30	109	933	78

Concernant les outils de développement et compilateurs, nous avons utilisé Visual Studio 2010 (compilateur Microsoft v10). Sur GPP, les implémentations OpenCL AMD 2.4 et Intel 1.1 ont été évaluées, ainsi que OpenMP 2.0 et nous avons fait l'usage des instructions SIMD SSE2 (uniquement pour le filtrage).

Les mesures de temps ont été effectuées à l'aide du compteur haute-résolution *QueryPerformanceCounter* pour le GPP. Les mesures GPU ont été réalisées à l'aide des *events* pour les API CUDA et OpenCL (qui disposent aussi d'une résolution élevée : 0,5 μs et 1 μs respectivement).

2.3.2 Cadre d'utilisation et temps de transferts

Comme on l'a vu dans le chapitre d'introduction, les GPU sont fortement pénalisés par les transferts sur le bus PCI-E. Dans notre étude, nous ne nous intéresserons pas à ceux-ci, dans la mesure où l'on souhaite que les données nécessaires à l'algorithme soient considérées comme résidentes en RAM GPU. L'idée est de les envoyer puis de les maintenir pour effectuer plusieurs reconstructions. Nous discuterons cette problématique plus tard dans le chapitre.

2.3.3 Présentation des jeux de données

Pour illustrer nos benchmarks, nous avons choisis deux jeux de données différents. Voici une présentation de leurs spécificités.

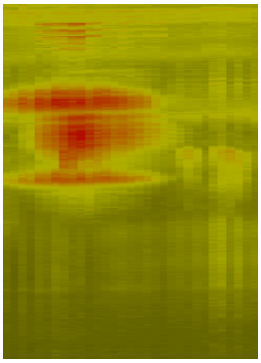
2.3.3.1 PMF

PMF est un jeu de données issu d'une acquisition dont les trajets ont systématiquement 2 segments. Il est composé de 150k tirs avec 1k échantillons par signal. Cela représente donc une reconstruction de 150M de projections. La figure 2.7 présente les différentes vues obtenues lors de la reconstruction de ce jeu de données. On constate sur la vue de côté que malgré la régularité des trajets, ils ne sont pas tous orientés de la même manière. Cela est dû à un balayage électronique. On va donc avoir une probabilité de collision plus importante que si les trajets étaient parfaitement réguliers et répartis.

Concernant la taille du traitement d'image, on a les valeurs suivantes pour les 3 dimensions :$1mm \times 0{,}5mm \times 32{,}3 \cdot 10{-}3mm$.

(b) Vue de côté

(a) Vue de dessus



(c) Vue de face

FIGURE 2.7 – Reconstructions du jeu de données PMF

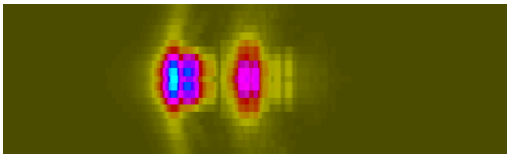
2.3.3.2 EXZ

EXZ est un jeu de données issu de la simulation d'un contrôle d'une soudure. On a donc une disparité notable au niveau du nombre de segments par trajets qui varie de 2 à 10. Le contrôle est composé de 15k tirs avec 2k échantillons par signal, ce qui représente donc une reconstruction de 30M de projections. La figure 2.8 présente les différentes vues obtenues lors des reconstructions. Sur la vue de côté, on constate que les trajets sont très peu réguliers. Etant donné que la pièce a été réalisée avec une extrusion 2D, les vues de dessus ou de face ne permettent pas de voir ces irrégularités, puisqu'elles ne sont visibles que dans le plan 2D associé à l'extrusion. En revanche, les irrégularités peuvent avoir un impact sur la quantité de collisions, et ce, même sur les vues de face et de dessus.

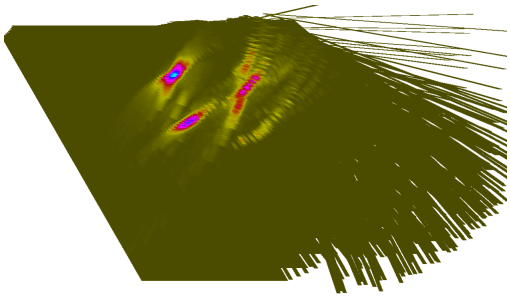
Concernant la taille du traitement d'image, on a les valeurs suivantes pour les 3 dimensions : $3mm \times 1{,}5mm \times 59 \cdot 10^{-3}mm$. La taille du traitement est donc bien plus importante pour ce jeu de données que pour PMF.

2.4 Mise en évidence des approches de parallélisation

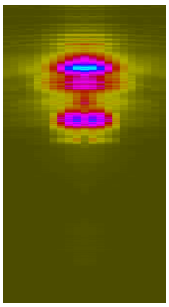
Nous avons identifié deux approches de parallélisation permettant d'obtenir un découpage des calculs adaptables à la fois sur GPP multicoeurs et GPU. Dans un premier temps, nous allons commencer par détailler l'approche de parallélisation sur les pixels, puis dans un second temps nous présenterons une seconde approche de parallélisation sur les trajets.



(a) Vue de dessus



(b) Vue de côté



(c) Vue de face

FIGURE 2.8 – Reconstructions du jeu de données EXZ

2.4.1 Approche par parallélisation pixel

A l'instar des algorithmes de *ray-tracing*, l'approche de parallélisation sur les pixels nécessite de traverser le volume de données. Dans notre cas, le volume est traversé de manière orthogonale au plan de l'image.

La principale difficulté qui se présente avec cette parallélisation est qu'il va être nécessaire de déterminer pour chaque pixel, les contributions qui lui sont propres. Pour ce faire, on utilise la projection orthogonale du pixel dans la pièce et on va effectuer un calcul d'intersection avec les différents segments de trajets. La figure 2.9 présente un schéma de ce calcul d'intersection. Nous allons donc avoir un très grand nombre de calculs d'intersection : pour une configuration effectuant N_t tirs et une image de N_p pixels, le nombre de calculs d'intersection va être en $O(k \times N_t \times N_p))$ avec k le nombre de segments par trajet.

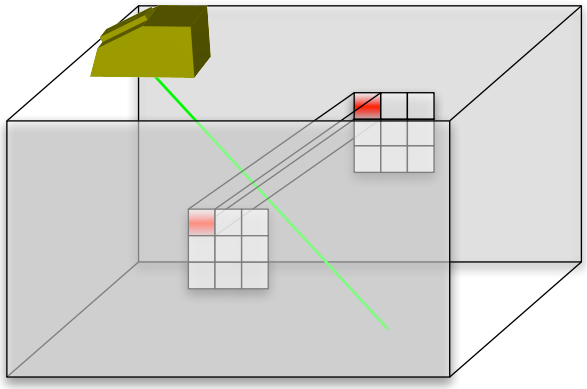


FIGURE 2.9 – Schéma représentant le calcul nécessaire pour effectuer une parallélisation sur les pixels. On effectue pour chaque pixel un calcul d'intersection entre la projection de ce pixel (ici en rouge), et le trajet (en vert).

Cette approche est donc très coûteuse du fait du produit entre les deux grandeurs les plus importantes de nos entrées : O($N_t \times N_p$). Réalisée en début de thèse, une maquette sur GPU de cette approche nous a confirmé les mauvaises performances que l'on pouvait attendre du fait de cette complexité algorithmique élevée. Nous nous sommes donc orientés sur une autre approche que nous allons maintenant détailler.

2.4.2 Approche par parallélisation trajet

L'approche de parallélisation sur les trajets est une parallélisation sur la boucle principale de l'implémentation de référence. En effet, chaque trajet est projeté sur l'image indépendamment des autres. De plus, tous les signaux sont de la même dimension. On a donc un nombre de projections équivalent entre chaque trajet. La seule chose qui peut changer est le découpage en segments, et donc la répartition des échantillons par segment.

A chaque trajet, on dispose en lecture d'informations propres : un signal et des coordonnées géométriques des points aux extrémités et interfaces. En revanche, en écriture, la structure des trajets fait que plusieurs trajets vont se projeter sur le même pixel. Pour cette raison, il faut trouver des moyens de gérer les accès mémoire concurrents.

Autant sur GPP, il semble possible de contourner le problème en créant autant d'images que l'on a de threads de calcul du fait du nombre relativement faible de ces derniers. Autant sur GPU, cette stratégie n'est pas envisageable. Nous allons devoir gérer la sérialisation des accès via un autre mécanisme : les instructions atomiques des GPU (c.f. chapitre 1). Ces instructions existent depuis la *compute capability 1.1* des architectures GPU compatibles CUDA. Ces instructions sont relativement coûteuses mais ont été améliorées de manière très significative depuis l'arrivée de l'architecture Fermi (*compute capability 2.x*). En effet, Nvidia parle d'améliorations pouvant aller jusqu'à un facteur $\times 20$ par rapport aux anciennes générations de GPU [Nvidia, 2009]. Nous pensons donc qu'il doit être possible de les employer tout en restant efficace.

Comme nous l'avons décrit dans le chapitre 1, pour paralléliser un code sur GPU, il faut s'assurer qu'il y ait assez de calculs à répartir sur le GPU. En effet, le GPU disposant de plusieurs centaines d'unités de calculs, il est nécessaire de lancer plusieurs milliers de threads en parallèle afin de permettre au GPU d'ordonnancer les threads de manière à recouvrir au maximum les accès mémoire et donc d'être plus efficace. Dans notre cas, les contrôles qui nous intéressent disposent au minimum d'une dizaine de milliers de trajets. On est donc dans un cas favorable quant aux données sur lesquelles on souhaite paralléliser l'algorithme.

2.5 Etude des performances de la parallélisation sur GPU

Dans cette section, nous allons commencer par présenter l'implémentation GPU. Nous montrerons pourquoi l'approche par parallélisation trajet est une approche exploitable pour le GPU. Ensuite nous passerons à l'aspect benchmark en comparant deux générations d'architectures GPU Nvidia ainsi que les deux modèles de programmations CUDA et OpenCL.

2.5.1 Implémentation GPU

Comme nous l'avons expliqué dans la section précédente, l'écriture des données va devoir être synchronisée avec une instruction atomique afin de sérialiser les possibles accès à un même pixel. Concernant ces instructions, deux aspects doivent être différenciés : la latence d'accès à la mémoire, et la latence due au verrou appliqué pour sérialiser l'instruction. La latence d'accès est donc uniquement due au coût d'accès à la mémoire globale (pour rappel, de l'ordre de plusieurs centaines de cycles mémoire). Quant à la latence du verrou, c'est le temps nécessaire à l'instruction pour prendre la main sur l'adresse mémoire, s'exécuter, puis rendre la main.

En pratique, les GPU sont capables de recouvrir une grande partie des latences mémoires s'il y a suffisamment de calculs. Pour cette raison, si les accès sont suffisamment espacés, la latence de verrou ne devrait donc pas être trop pénalisante, pourvu qu'on ait assez de calculs. En revanche, lorsque le nombre de collisions est important, les temps de latence des instructions s'ajoutent, ce qui devient très pénalisant.

La figure 2.10 présente la parallélisation par un schéma. Au niveau implémentation, par rapport à l'algorithme 1, la boucle supérieure a été retirée et on a placé un calcul de trajet par thread GPU. Une seule image de sortie est utilisée et chaque thread synchronise ses accès à l'aide d'un `atomicMax` (instructions garantissant les accès à la mémoire en exclusion mutuelle, cf section 1.2.2.2).

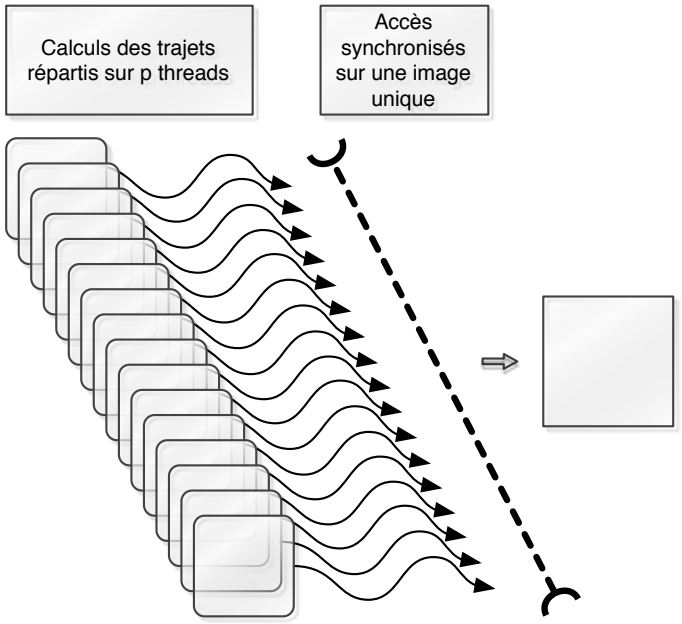


FIGURE 2.10 – Schéma de la parallélisation GPU pour l'algorithme IVC. Chaque carré à coin arrondi représente une tâche de calcul correspondant à un couple trajet/signal. Tous les accès à l'image se font de manière synchrone via une instruction atomique du GPU.

Nous avons implémenté deux versions de l'algorithme IVC. La première se base sur l'algorithme de base, incluant le traitement d'image à la volée et sera nommé dans la suite GPU-A. La seconde implémentation n'applique pas le traitement à la volée mais en post-traitement, tel que nous l'avons présenté précédemment, et sera nommée GPU-B. L'implémentation réalisée pour la dilatation s'inspire du code de calcul de convolution séparable du SDK de CUDA [Podlozhnyuk, 2007].

2.5.2 Validation de l'approche de parallélisation sur GPU

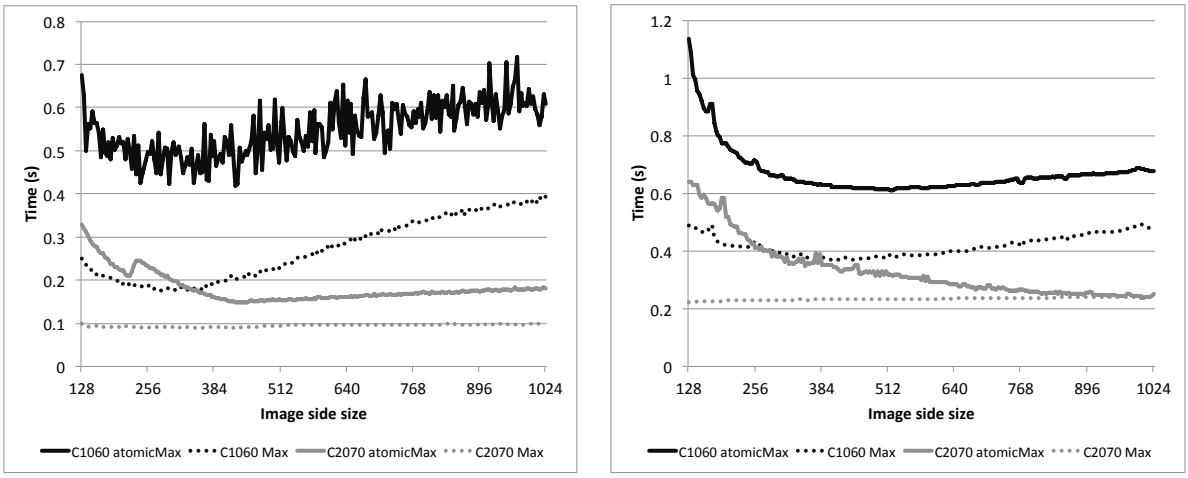
Comme nous l'avons vu dans la section 2.4.2, l'approche de parallélisation choisie oblige à passer par des instructions atomiques sur GPU. Pour cela, nous nous sommes intéressés à leurs performances. Nous avons donc effectué des tests unitaires sur les deux générations d'architectures Nvidia puis une étude de la répartition des accès mémoire sur l'image pour l'utilisation du kernel GPU-B sur l'architecture Fermi.

2.5.2.1 Etude de performances des instructions atomiques du GPU

Etant donné l'utilisation intensive des instruction atomiques, nous nous sommes intéressés à leur coût. Pour cela nous avons réalisé deux tests unitaires. Tout d'abord, nous avons comparé le coût d'un `atomicMax` par rapport à un calcul du max classique, pour des accès pseudo-aléatoires sur une zone linéaire (que l'on peut comparer à notre image), pour laquelle on va faire varier la taille. Nous ignorons ici le fait que le max n'assure pas la sérialisation des accès et par conséquent la validité des résultats, mais c'est uniquement dans un but de comparaison de coût. Ensuite, nous avons réalisé cette même opération, non plus pour des accès pseudo-aléatoires, mais pour des accès correspondant à la projection de PMF pour des vues de côté. Nous avons effectué cette étude sur deux générations de GPU Nvidia en utilisant les Tesla C1060 et C2070.

Il faut savoir que sur la carte C1060, la mémoire globale n'est pas cachée [Wong et al., 2010] tandis que la C2070 dispose de deux niveaux de cache. Le cache L1 est local au SM tandis que le L2 est global à tous les SM. Le cache L1 ne sera utile qu'en lecture, même si en écriture, les instructions atomiques peuvent effectuer une réduction dans ce cache avant de repasser par le L2 (le contrôleur mémoire dédié aux instructions atomiques étant situé au niveau du L2). Etant donné qu'on dispose d'une quantité de calculs relativement peu élevée par rapport aux accès mémoire, on se pose donc la question des performances de ces instructions, et particulièrement sur l'architecture Fermi.

La figure 2.11 (a) présente le résultat pour des accès pseudo-aléatoires pour un calcul de max et d'`atomicMax` sur C1060 et C2070. 150M points sont écrits dans une zone dont la taille varie entre 128×128 et 1024×1024 (valeurs choisies en fonction de la taille qui va nous intéresser lors de l'intégration de l'algorithme). L'augmentation de la taille de l'image permet de réduire le nombre de collisions par pixels pour un ensemble de tirs donnés.



(a) Accès pseudo-aléatoires (b) IVC

FIGURE 2.11 – Comparaison de Tesla C1060 et C2070 pour des calculs de max standard et atomique sur une taille d'image variant entre 128×128 à 1024×1024. (a) Accès pseudo-aléatoires. (b) Accès correspondant à une projection de côté sur le jeu de données PMF.

Sur C1060, on constate que la courbe pour les `atomicMax` est très bruitée comparée à celle de la C2070. Si l'on compare par rapport a un max, l'`atomicMax` est environ $\times 2{,}5$ plus lent. Sur C2070, la synchronisation a bien-sûr un coût important et particulièrement lorsque l'image est petite et donc les accès concurrents importants. En revanche, l'écart entre les deux courbes se resserre pour atteindre 75% des performances du max en 350×350 ce qui implique que la latence des instructions s'est grandement améliorée entre les deux générations de GPU.

La figure 7 (b) présente donc une projection correspondant à une vue de côté pour le jeu de données PMF. On constate que pour des images de taille supérieure à 192×192, les courbes pour la C1060 sont écartées d'un facteur $\times 1{,}5$ voulant dire que le coût du verrou de l'instruction est non significatif car on n'a peu de concurrence. On constate que ce coût apparait pour des images de taille inférieure allant jusqu'à un facteur $\times 3$ pour des images de 128×128. Cela montre à quel point les instructions atomiques peuvent être coûteuses sur cette génération de GPU.

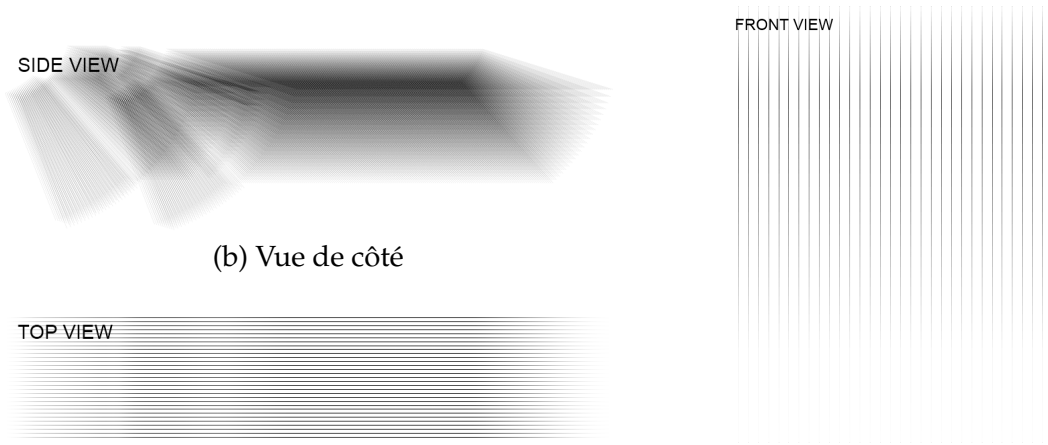
Pour la C2070, on constate que l'allure des courbes est très différente de celle de la C1060. La courbe du max est très stable avec une légère augmentation de l'ordre de 10% entre des images de 128×128 et 1024×1024. La différence principale réside dans le fait que la courbe avec `atomicMax` décroit avec l'augmentation de taille de zone et parvient même à rejoindre la courbe du max. Cela s'explique par le fait que les accès mémoire avec le max passent par des accès cache de 128 octets, alors qu'avec l'`atomicMax`, ces accès sont hors cache L1, et par conséquent uniquement réalisés sur 32 octets. On peut malgré cela constater que les performances des instructions atomiques sur l'architecture Fermi sont plus efficaces. De plus, nos courbes montrent un gain d'un facteur $\times 1{,}5$ à $\times 2{,}8$ pour des calculs réalisés en 1024×1024.

Pour l'algorithme IVC, l'utilisation d'un `atomicMax` est nécessaire pour assurer un calcul correct du maximum. Cela dit, le coût d'utilisation de ces instructions peut devenir non significatif sur l'architecture Fermi. Cela nécessite une taille d'image suffisamment grande pour que les accès concurrents soient réduits au minimum, mais cela permet de nous rendre compte que ces instructions sont tout à fait exploitables, même de manière intensive comme pour PMF ou l'on projette 150 millions de points (soit 150 millions d'`atomicMax`).

On va donc maintenant s'intéresser à la répartition des accès mémoire pour le jeu de données PMF afin de savoir si la concurrence va nous pénaliser et à quel point.

2.5.2.2 Ordonnancement des accès mémoire en cas de parallélisation trajet

Afin d'obtenir une meilleure vue des possibilités de collisions et de mieux comprendre les différences de temps de calcul entre les vues (que nous verrons par la suite, nous avons pris le jeu de données PMF pour faire cette étude et utilisé l'architecture Fermi puisque nous avons montré que l'ancienne génération était très en retrait en termes de performances sur l'utilisation des instructions atomiques.



(b) Vue de côté

(a) Vue de dessus

(c) Vue de face

FIGURE 2.12 – Histogrammes des accès mémoire pour les différentes vues du jeu de données PMF pour le kernel GPU-B.

La figure 2.12 présente des histogrammes des accès à la mémoire lors de l'application du kernel GPU-B. On peut immédiatement voir que les accès vont être très différents selon les vues : on constate bien le caractère ponctuel des projections d'amplitudes. A priori, sur les vues de face et de dessus, on devrait avoir une quantité de collisions bien plus importante que pour la vue de côté. Sur la vue de côté, certains pixels sont accédés jusqu'à 3000 fois, jusqu'à 9000 fois pour la vue de dessus, et 14000 fois sur la vue de face. Ceci est très intéressant car même si ce nombre peut paraitre extrêmement élevé, cet ensemble d'accès est réparti sur toute l'exécution du kernel, ce qui rend les collisions bien moins importantes qu'il peut paraitre.

Tableau 2.3 – Informations issues du profiling du kernel GPU-B sud C2070.

Vue	Temps (ms)	Transactions mémoire par requête
Vue de côté	592	26,4
Vue de dessus	457	4,4
Vue de face	628	31,0
Vue de face (90°)	489	5,8

Observons le tableau 2.3. Le temps de calcul pour la vue de dessus est le plus faible. Cela s'explique par l'alignement des trajets des tirs d'un incrément avec l'alignement mémoire du stockage de l'image. En effet, pour les vues de côté et de face, cet alignement n'existe pas. Cela explique donc les temps de calcul. Nous remarquons aussi que lorsque l'alignement n'existe pas, le nombre de transactions mémoires par requête est beaucoup plus important. En dernière ligne du tableau se trouve une vue de face dont on a transposé l'image de manière à obtenir des accès alignés tels que ceux de la vue de dessus. On constate une nette amélioration des performances avec une réduction très importante des transactions par requête. On obtient donc une valeur très proche de celle de la vue de dessus.

La différence de performances entre ces deux vues n'est pas évidente car il s'agit principalement de la manière dont sont organisés les trajets. Parmi les différentes valeurs que le profiler nous donne, nous pouvons noter le *Warp Issue Efficiency* qui montre que les ordonnanceurs de warps sont deux fois plus souvent en attente d'un warp pour la vue de face que pour la vue de dessus. Cela montre que la vue de face est handicapée par des latences mémoires plus importantes.

Pour la suite de nos benchmarks, nous utiliserons la vue de face non transposée dans la mesure où de manière générale, nous ne pouvons difficilement prévoir les possibles fortes irrégularités entre les trajets.

2.5.3 Benchmarks et analyses de performances

Nous allons commencer par présenter l'impact de l'optimisation du traitement d'image vu en section 2.2 . Ensuite, nous nous intéresserons à l'impact de l'irrégularité des trajets. Nous comparerons les performances de l'architecture Fermi pour les différents modes de caches. Enfin, nous terminerons par une comparaison opposant CUDA à OpenCL.

2.5.3.1 Impact de l'optimisation du traitement d'image

Nous souhaitons ici comparer le coût de la dilatation entre l'implémentation GPU-A et GPU-B. Pour cela, nous choisissons un sous-ensemble de nos architectures et modèles de programmation. Nous allons comparer nos deux Tesla sur les implémentations GPU-A et GPU-B sur les jeux de données EXZ et PMF. Le but ici est de valider l'optimisation du traitement (vu en section 2.2 dont on a parlé précédemment et de chiffrer le gain obtenu sur GPU.

Tableau 2.4 – Présentation des gains grâce à l'optimisation algorithmique du traitement d'image, ainsi que des gains entre les deux générations d'architectures GPU Nvidia.

Vue	GPU-A		GPU-B		GPU-A / GPU-B		C1060 / C2070	
	C1060	C2070	C1060	C2070	C1060	C2070	GPU-A	GPU-B
	Jeu de données EXZ - reconstruction 800 x 800							
Vue de côté	0,92	0,34	0,14	0,072	x6,6	x4,7	x2,7	x1,9
Vue de dessus	10,36	1,71	0,21	0,068	x49,3	x25,1	x6,1	x3,1
Vue de face	6,19	1,86	0,24	0,077	x25,8	x24,2	x3,3	x3,1
	Jeu de données PMF - reconstruction 800 x 800							
Vue de côté	2,4	1,3	0,7	0,26	x3,4	x5,0	x1,8	x2,7
Vue de dessus	10,8	2,8	0,8	0,11	x13,5	x25,5	x3,9	x7,3
Vue de face	20,0	11,1	1,0	0,29	x20,0	x38,3	x1,8	x3,4

Le tableau 2.4 nous permet d'avoir une vue d'ensemble des gains au niveau algorithmique mais aussi au niveau des deux générations d'architectures. On peut constater que les gains ne sont pas les mêmes pour les deux jeux de données. En effet, pour EXZ, la taille du traitement d'image est de $3mm \times 1{,}5mm$ au maximum (pour la vue de dessus). C'est aussi sur cette vue qu'on constate le gain au niveau algorithmique le plus important (GPU-A / GPU -B dans le tableau). Cela s'explique par cette taille de traitement, qui pour ce GPU implique une localité très forte au niveau des accès atomiques.

Les gains au niveau de l'architecture (C1060 / C2070 dans le tableau) montrent que la C1060 a beaucoup plus à gagner sur l'implémentation GPU-A. Cela s'explique simplement par le fait que la C1060 est plus handicapée que la C2070 lors de l'usage de ces instructions. Lorsque l'intensité est moins forte, que les accès concurrents sont moins nombreux comme pour l'implémentation GPU-B, ces gains entre architectures sont réduits.

La régularité des trajets du jeu de données PMF permet à la C2070 de profiter de plus d'accès coalescents, et donc de meilleures performances pour l'implémentation GPU-B. Nous allons maintenant comparer les performances entre les deux jeux de données et nous intéresser plus particulièrement à l'impact de l'irrégularité des trajets.

2.5.3.2 Impact de l'irrégularité des trajets

Afin de comparer ces informations, on peut reprendre les informations du tableau 2.4 en s'intéressant plus particulièrement à l'implémentation GPU-B. Cette implémentation est sensée être de complexité $O(N_T \times N_E)$. Nous allons vérifier cela en normalisant les temps de calculs par le nombre de projections.

Le tableau 2.5 présente des résultats normalisés en fonction du nombre de projections liées au jeu de données (30 millions pour EXZ, 150 millions pour PMF). On se base sur l'hypothèse que les contrôles sont réalisés selon les mêmes axes et donc que les vues sont comparables une à une. On constate que sur C1060, les gains lorsque l'on passe sur un jeu de de données plus régulier tel que PMF vont jusqu'à 30%. En revanche, pour la vue de côté, on ne constate aucun gain. La raison est simplement que la concurrence est très faible et que par conséquent, on est limité par les accès à la mémoire globale du GPU.

Tableau 2.5 – Temps de calculs normalisés en fonction du nombre d'échantillons à projeter pour l'algorithme GPU-B de IVC. Les facteurs d'accélérations montrent le gain entre l'exécution sur un jeu régulier comme PMF et un irrégulier comme EXZ.

	EXZ		PMF		Gain EXZ/PMF	
	C1060	C2070	C1060	C2070	C1060	C2070
Vue de côté	$4{,}7 \cdot 10^{-9}$	$2{,}4 \cdot 10^{-9}$	$4{,}7 \cdot 10^{-9}$	$1{,}7 \cdot 10^{-9}$	$\times 1{,}0$	$\times 1{,}4$
Vue de dessus	$7{,}0 \cdot 10^{-9}$	$2{,}3 \cdot 10^{-9}$	$5{,}3 \cdot 10^{-9}$	$7{,}3 \cdot 10^{-10}$	$\times 1{,}3$	$\times 3{,}1$
Vue de face	$8{,}0 \cdot 10^{-9}$	$2{,}6 \cdot 10^{-9}$	$6{,}7 \cdot 10^{-9}$	$1{,}9 \cdot 10^{-9}$	$\times 1{,}2$	$\times 1{,}3$

En revanche, sur C2070, on constate un gain systématique, qui va de 30% pour la vue de côté à 200% pour la vue de dessus. En effet, la vue de dessus profite beaucoup plus à l'architecture Fermi de par le passage en cache L2 des accès atomiques. On ne peut donc que constater la perte de performance engendrée par des trajets irréguliers et désorganisés tel que dans le cas du jeu de données EXZ.

2.5.3.3 Fermi : modes de cache et comparaison C2070/GTX580

Sur l'architecture Fermi, de *compute capability 2.x* ou plus, il est possible au moment de la compilation de modifier les paramètres des caches. Quatre paramètres sont possibles :

- *ca* : mode par défaut, cache L1 et L2 actifs
- *cg* : cache L1 désactivé, cache L2 actif.
- *cv* : cache L1 désactivé, cache L2 normalement accédé une à deux fois maximum.
- *cs* : cache considéré comme volatile, donc relecture en mémoire globale obligatoire.

On s'intéresse plus particulièrement à la désactivation du cache L1, puisque dans notre cas, toutes nos écritures passent par le L2 directement. On n'a donc uniquement que les lectures qui peuvent bénéficier du cache L1. En désactivant le cache, on passe d'accès de 128 octets en mémoire globale à des accès de 32 octets. Si le cache est mal utilisé, on devrait observer une amélioration des temps de calcul.

Tableau 2.6 – Comparaison entre C2070 et GTX580 et impact de la désactivation du cache L1 sur cette dernière (reconstruction PMF en 800×800).

Vue	Côté	Dessus	Face
GPU	PMF - 800x800 - Temps (ms)		
C2070 L1 ON	257,8	109,7	295,6
GTX580 L1 ON	194,4	90,6	222,3
GTX580 L1 OFF	192,3	76,8	222,4

Nous avons effectué des tests sur la GTX580 et comparé les résultats à la C2070 sur le jeu de données PMF pour des reconstructions en 800×800. Le tableau 2.6 présente les résultats obtenus. On constate que la désactivation du cache ne permet de gagner que pour la vue de dessus. Cela s'explique par le fait que seule cette vue profite d'accès alignés, et par conséquent bénéficie du cache L2 sans pour autant être limitée par les accès atomiques.

Concernant la différence entre la C2070 et la GTX580, on a pu voir dans les spécifications que 250MHz (1002MHz pour la GTX580 contre 747MHz pour la C2070) séparent ces deux GPU au niveau mémoire. Sachant que notre algorithme GPU-B est limité par la bande passante, on pourrait s'attendre à obtenir une différence entre les deux GPU d'environ 25%. On constate que pour les vues de côté et de face, on obtient un gain de l'ordre de 30% ce qui montre que la bande passante n'est pas seule responsable du gain, mais bien aussi les unités de calcul supplémentaires de la GTX580. En revanche, pour la vue de dessus, on constate que le gain n'est que de l'ordre de 10%, valeur que nous observons, mais pour laquelle nous n'avons pas d'explication.

2.5.3.4 Comparaison CUDA / OpenCL

On s'est aussi intéressé à l'implémentation de l'algorithme GPU-B en OpenCL. Nous avons porté le code CUDA directement en OpenCL. Nous avons souhaité évaluer les performances de ce modèles sur nos architectures Fermi. Comme nous l'avons dit dans le chapitre 1, OpenCL dispose d'un modèle qui nécessite une compilation au *runtime*. Dans nos benchmarks, nous ne prenons pas en compte les temps de compilation, dans la mesure où nous nous basons sur une utilisation d'un fichier binaire précompilé et dont la compilation se fait dans un temps inférieur à 3 ms.

Le désavantage du fichier précompilé est qu'il l'est pour une plateforme donnée. En effet, il sera impossible d'utiliser un binaire précompilé pour GPU Nvidia sur un GPU AMD, ou un même un GPP. Cependant, il se trouve que la compilation depuis un fichier source nous a posé différents problèmes : Nvidia propose un système de cache des binaires compilés pour accélérer le temps de compilation si le fichier source n'a pas changé. Ce n'est pas le cas pour les implémentations AMD et Intel. Le temps de compilation étant très important depuis un source pour ceux-ci, nous avons uniformisé en utilisant des binaires précompilés.

Tableau 2.7 – Comparaison CUDA / OpenCL sur les deux GPU Fermi (reconstruction PMF en 800×800).

	CUDA			OpenCL		
Vue	Côté	Dessus	Face	Côté	Dessus	Face
GPU	PMF - 800x800 - Temps (ms)					
C2070 L1 ON	257,8	109,7	295,6	261,9	109,6	294,3
GTX580 L1 ON	194,4	90,6	222,3	194,1	90,2	221,4
GTX580 L1 OFF	192,3	76,8	222,4	N/A	N/A	N/A

Le tableau 2.7 reprend les valeurs de la sous-section précédente et ajoute les temps de calcul OpenCL pour la C2070 et la GTX580. Sachant que les résultats sur le jeu de données EXZ n'apporte rien de plus que ceux sur PMF, nous ne les avons pas inclus dans ce tableau. Sachant qu'OpenCL ne permet pas de jouer sur les modes de cache des Fermi, nous n'avons pas de valeurs (N/A dans le tableau). En revanche, on peut constater la très bonne tenue des deux Fermi par rapport à l'implémentation CUDA. On constate même des temps de calculs très légèrement inférieurs du côté d'OpenCL. Sachant que le système de *timers* n'est pas le même pour les deux API, on peut considérer que l'exécution via OpenCL est aussi véloce que celle de CUDA.

2.6 Etude de la parallélisation sur GPP multicoeurs

2.6.1 Implémentations GPP multicoeurs

La parallélisation pour GPP multicoeurs est, comme nous l'avons mentionné précédemment, basée sur un partage des calculs de trajets. Chaque couple trajet/signal est indépendant des autres couples, du moment que les threads de calcul n'écrivent pas de manière simultanée sur la même image. Pour cela, on va utiliser une image différente par threads, puis les fusionner par réduction une fois les calculs de projection terminés.

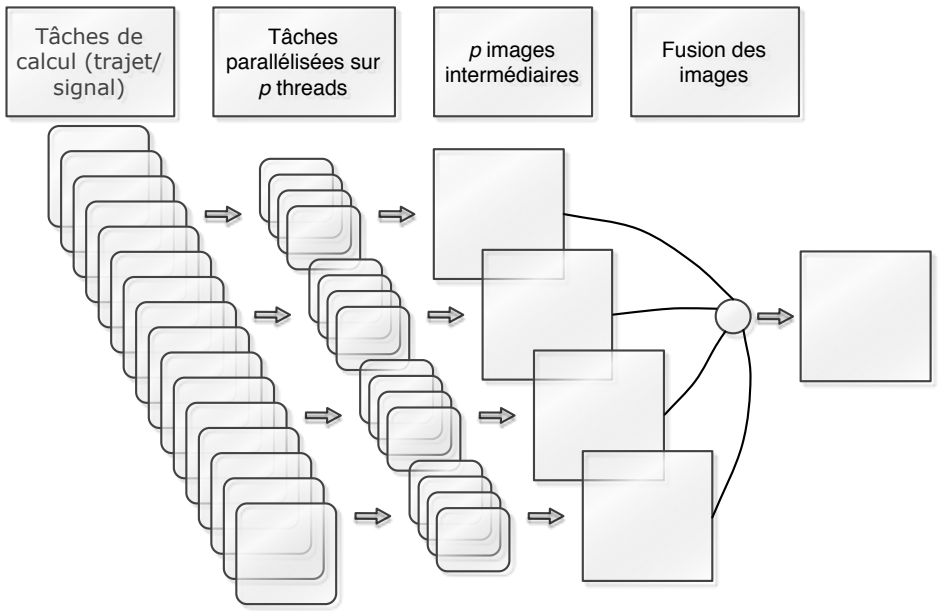


FIGURE 2.13 – Schéma de la parallélisation GPP pour l'algorithme IVC. Chaque carré à coin arrondi représente une tâche de calcul correspondant à un couple trajet/signal. Chaque thread dispose d'une image privée pour éviter la synchronisation pendant les calculs. Une étape de fusion de ces images est nécessaire une fois les projections effectuées.

Une autre méthode de parallélisation serait d'utiliser une section critique, à l'aide par exemple d'OpenMP et du mot clé *critical*. Nous avons écarté cette possibilité car l'espace mémoire de l'image est complètement bloqué ; contrairement aux instructions atomiques des GPU qui ne bloquent l'accès qu'à l'espace mémoire accédé.

La figure 2.13 présente cette parallélisation. L'étape de fusion a été parallélisée et nous avons utilisé des instructions SIMD SSE pour accélérer le calcul. La boucle principale sur les trajets a été parallélisée à l'aide d'OpenMP. Les images intermédiaires ont été pré-allouées avant la boucle afin d'éviter une allocation dynamique dans les sections parallèles. Le reste des variables est privée, ou bien partagée, mais uniquement en lecture. Cela nous permet d'avoir une implémentation la plus efficace possible.

OpenCL a aussi été évalué à l'aide de l'implémentation réalisée sur GPU. Le portage n'a nécessité aucun travail supplémentaire, dans la mesure où nous souhaitions tester l'adaptation d'un code développé pour le GPU sur GPP multicoeurs. Il est à noter que n'ayant pas nativement d'instruction atomiques, les GPP utilisent l'instruction x86 *lock*. Ce type d'opération fait l'objet d'un intérêt particulier chez Intel puisque leur future architecture, Haswell, disposera d'une mémoire transactionnelle permettant d'accélérer ce type d'opérations.

2.6.2 Benchmarks et analyses de performances

Nous avons réalisé différents benchmarks pour vérifier l'efficacité de la parallélisation GPP multicoeurs via OpenMP et OpenCL. Nous commencerons par donner des résultats de passage à l'échelle d'OpenMP sur nos deux générations de GPP. Puis en second nous comparerons les résultats OpenMP, OpenCL AMD et OpenCL Intel.

2.6.2.1 Impact du passage en post-traitement du traitement d'images

Nous avons comparé le coût du traitement d'image entre l'implémentation GPP- A et GPP-B. Nous avons effectué ces tests sur notre quad-coeur pour les jeux de données EXZ et PMF. Le but est ici de valider l'optimisation du traitement d'image sur GPP.

Tableau 2.8 – Gains obtenus grâce à l'optimisation du traitement d'images sur GPP.

	GPP-A	GPP-B	GPP-A / GPP-B
	EXZ - 800x800 - Temps (s)		
Côté	5,02	1,16	$\times 4$
Dessus	43,12	1,19	$\times 36$
Face	14,07	1,22	$\times 12$
	PMF - 800x800 - Temps (s)		
Côté	18,6	4,19	$\times 4$
Dessus	60,3	4,29	$\times 14$
Face	62,2	4,30	$\times 14$

Le tableau 2.8 présente les gains GPP entre l'algorithme GPP-A et GPP-B pour l'implémentation GPP monothreadée. On constate comme sur GPU des gains très importants dépendants de la taille du traitement et donc dépendant du jeu de données. Si on normalise le temps de calcul par le nombre de projections on obtient : pour EXZ, $4{,}0 \cdot 10^{-8}$ secondes par projection, et pour PMF, $2{,}8 \cdot 10^{-8}$ secondes par projection (en cycle d'horloge, cela représente respectivement 120 et 84 cycles par projection). On peut s'apercevoir que l'on se rapproche d'une implémentation dont le temps de calcul est prédictible. La différence qui existe entre les deux jeux de données et due à l'irrégularité des trajets du jeu de données EXZ.

2.6.2.2 Passage à l'échelle de l'implémentation OpenMP

On souhaite maintenant s'intéresser à la parallélisation en elle même et plus particulièrement à l'implémentation OpenMP. Le tableau 2.9 présente les résultats de parallélisation pour le jeu de données PMF sur les deux GPP testés.

Tableau 2.9 – Passage à l'échelle de l'implémentation OpenMP pour la projection de côté de PMF.

	Temps (s)				Gains		
Nb threads / Ratios	1	4	12	24	4 : 1	12 : 1	24 : 1
E5472	4296	1105	-	-	$\times 3{,}9$	-	-
X5690	2705	680	267	169	$\times 4{,}0$	$\times 10{,}1$	$\times 16{,}0$

On peut constater que les deux GPP passent presque parfaitement à l'échelle. Sur le X5690, on constate qu'avec 12 threads, on obtient un facteur d'accélération d'environ $\times 10$. Avec 24 threads, l'*Hyperthreading* permet d'obtenir un facteur d'accélération égal à $\times 16$. On constate ici que l'*Hyperthreading* permet d'augmenter de 30% les performances que l'on pourrait attendre du fait du nombre de coeurs. Cette valeur n'est pas surprenante par rapport à ce qui est présenté dans la littérature et montre que les changements de contextes permettent de mieux remplir le pipeline et donc d'obtenir une meilleure efficacité de calcul des coeurs du GPP.

2.6.2.3 Comparatif OpenMP / OpenCL

Le tableau 2.10 présente un comparatif des temps de calculs entre OpenMP, OpenCL Intel et OpenCL AMD pour l'algorithme GPP-B. On présente ici les meilleurs résultats obtenus pour OpenCL en ayant fait varier les dimensionnements de *workgroup* (cf. présentation OpenCL en section 1.3.2.1).

On constate ici que les performances atteintes par OpenCL sont de l'ordre de 20% inférieures à celles d'OpenMP. Il faut rappeler ici qu'OpenMP n'a pas de gestion de concurrence à la volée comme l'implémentation OpenCL. Malgré cette différence de performances, on peut constater un gain par rapport à l'implémentation monothreadée de l'ordre d'un facteur $\times 12$, ce qui, malgré tout, reste très intéressant.

La raison principale de cet intérêt est l'absence de modification du code développé pour GPU, qui s'adapte de manière efficace, sans aucun travail particulier au GPP multicoeurs. Il n'est pas surprenant qu'OpenCL ne puisse pas toujours être aussi performant que les modèles de programmation natifs, mais dans le cas de l'algorithme IVC, OpenCL est une alternative

Tableau 2.10 – Temps de calcul en secondes sur X5690 pour l'algorithme GPP-B sur OpenMP, OpenCL AMD et OpenCL Intel

Vue	Côté	Dessus	Face
OpenMP	169.3	172.0	170.7
OpenCL AMD	215.3	220.0	217.1
OpenCL Intel	217.3	214.7	218.3
OpenCL AMD / OpenMP	-21%	-22%	-21%
OpenCL Intel / OpenMP	-22%	-20%	-22%

très intéressante.

Pour terminer cette partie, on peut aussi noter qu'entre OpenCL Intel et OpenCL AMD, les performances sont du même ordre de grandeur. Nous allons maintenant passer au comparatif GPP/GPU.

2.7 Analyse générale GPP multicoeurs et GPU

Dans cette section, nous allons regrouper les résultats précédents pour tirer des conclusions sur la parallélisation de l'algorithme IVC sur GPP multicoeurs et GPU sur différents modèles de programmation.

2.7.1 Comparaison GPP / GPU

Au moment où nous avons effectué les benchmarks, nous avions une machine disposant du processeur le plus performant (nombre de coeurs et fréquence) du marché, et des GPU les plus performants eux aussi. Nous pouvions donc à cet instant, comparer les performances des deux architectures de manière équitable de notre point de vue. D'un côté, un double Xeon 6 coeurs X5690, et de l'autre une Tesla C2070 et une GeForce GTX 580, les architectures GPU les plus performantes chez Nvidia.

Le tableau 2.11 présente un récapitulatif des performances obtenues sur GPP et GPU. Etant donné les différences de performances obtenues entre les différentes vues sur GPU, les différences entre GPP et GPU sont variables. Sur la vue de dessus, qui profite des caches GPU, on peut observer un gain de performances de 1,6 par rapport au GPP parallélisé sur 24 threads (par rapport à la GTX 580, sans cache L1). En revanche, on constate l'effet inverse là où les accès sont trop aléatoires et lorsque les instructions atomiques deviennent coûteuses. Dans ce cas, le GPP repasse devant pour les vues de face avec jusqu'à un facteur $\times 2$ d'écart.

Tableau 2.11 – Récapitulatif des performances obtenues pour l'algorithme IVC.

Architecture	Langage/Outil	Vue de côté	Vue de dessus	Vue de face
X5690 (2)	Séquentiel	2708	2731	2713
	OpenMP (24)	169	172	171
	OpenCL AMD	215	220	217
	OpenCL Intel	217	214	218
GTX 580	CUDA (L1 OFF)	192	77	222
	OpenCL	194	90	221
C2070	CUDA	258	110	296
	OpenCL	262	110	294

Les performances sont donc variables selon les cas, mais on peut globalement constater que le bi-processeur Intel avec *Hyperthreading* est à un niveau de performances équivalent aux GPU de dernière génération tout en étant aussi bien plus stable.

2.7.2 Comparaison entre modèles de programmation

Nous avons comparé les architectures GPP et GPU entre elles, au travers d'implémentations réalisées avec leurs outils natifs. De plus, nous avons évalué leurs performances via une API hybride, OpenCL, au travers des implémentations Nvidia, AMD et Intel. Nous avons constaté que les performances d'OpenCL sur GPP étaient tout à fait exploitables, malgré une différence notable par rapport à OpenMP. Etant donné que le code n'était pas identique, il est difficile de tirer plus de conclusions sur l'efficacité d'OpenCL sur GPP multicoeurs. En revanche, les performances atteintes paraissent tout à fait intéressantes en vue d'un portage avec un code unique dans une plateforme industrielle.

L'idée d'avoir un code unique à exécuter sur différentes architectures parait être une solution idéale en termes de coût de développement et de maintenance. L'évolution du code, ou la correction de bogues multiplie les problèmes et le besoin de compétences multiples dans les équipes de développement, plus notamment d'un expert par architecture. On constate, que ce soit sur GPP ou GPU, une perte de performances allant jusqu'à 30% pour cet algorithme. Cela peut paraitre important, mais on peut considérer ce coût comme celui de la portabilité.

OpenCL est encore jeune, et nous avons évidemment eu différents problèmes lors du développement ; particulièrement au niveau des outils de debogage inexistants. Intel a récemment mis à disposition un outil, mais uniquement disponible pour Visual Studio 2008. AMD de son côté dispose d'un *debugger* uniquement accessible sur ses GPU. Quant à Nvidia, leur environnement de développement pour CUDA n'est pas disponible pour OpenCL. Malgré le fait d'avoir un code unique à développer, certains problèmes, comme le temps de compilation, émergent sur une plateforme et pas l'autre, ce qui rend plus complexe les développements.

Concernant les modèles natifs, la parallélisation OpenMP s'est bien comportée, avec des gains surlinéaires par rapport au nombre de coeurs (dû à l'*Hyperthreading*). Quant à CUDA, les outils de développement ont permis d'obtenir des informations précieuses lors des développements, ce qui s'avère être un atout important.

On considère donc pour l'algorithme IVC qu'OpenCL est une solution viable, à la fois sur GPP et sur GPU. Il semble tout de même plus prudent de débuter par un développement OpenMP qui nécessite moins de modifications de code. Le temps de développement de l'implémentation CUDA et du portage OpenCL auront été très importants par rapport au portage OpenMP. Disposant du code OpenCL, nous avons entrepris d'intégrer un prototype de l'algorithme IVC dans la plateforme CIVA.

2.8 Intégration de l'algorithme IVC dans la plateforme CIVA

L'intégration d'un prototype de l'algorithme IVC dans la plateforme CIVA a été réalisée afin d'observer les gains de performances pour l'utilisateur final, ainsi que les possibles modifications d'ergonomie. L'implémentation OpenCL a été choisie pour être intégrée avec l'idée de pouvoir rester au maximum indépendant de l'architecture cible, quitte à payer un certain prix niveau performances et fonctionnalités.

2.8.1 Présentation du prototype

L'intégration de l'algorithme TCV a nécessité une phase d'ingénierie en amont pour intégrer à la plateforme les accélérateurs OpenCL. Un ensemble de classes a été développé pour gérer la partie hôte de manière objet. Etant donné que les API de lecture des données sont développées en JAVA, il nous a paru plus pertinent de placer l'intelligence dans la même partie du logiciel. Nous avons utilisé un *wrapper* JAVA à OpenCL : JOCL [Hutter, 2012].

Par rapport à l'étude réalisée dans ce chapitre, les besoins de l'algorithme ont évolué au niveau du remplissage de l'image. En effet, le remplissage existant auparavant pouvait être amélioré en utilisant des informations supplémentaires : l'angle du trajet dont l'amplitude a été projeté, la profondeur de cette amplitude dans la pièce, le temps de vol et la direction du segment de trajet depuis lequel l'amplitude a été projetée. L'implémentation OpenCL (GPU et GPP) fait appel à un *atomic_max* exécutant un verrou sur une seule instruction. Le problème ici est qu'en plus de l'amplitude, on voudrait récupérer les informations citées ci-dessus. Il n'est pas possible d'effectuer plusieurs opérations atomiques de manière atomique.

Pour rappel, nous ne prendrons pas en compte la problématique des transferts de données. En effet, le choix a été fait d'effectuer un transfert total des données pour ensuite exécuter de multiples reconstructions sur ce jeu de données (cf. section 2.3.2). Le transfert est effectué manuellement par l'utilisateur qui spécifie quel jeu de données il va vouloir traiter grâce à l'accélérateur.

2.8.1.1 Contournement de la limitation de l'instruction atomique

Nous avons la possibilité d'utiliser l'opération atomique pour passer plusieurs informations. Celle-ci est effectuée sur 32 bits puisque les opérations atomiques en 64 bits ne sont disponibles qu'avec OpenCL 1.2 et que l'implémentation Nvidia est encore en 1.1. En l'occurrence, le facteur commun permettant de retrouver les informations citées ci-dessus étant les trajets, nous allons sortir en plus de l'amplitude, l'indice de trajet.

Pour se faire, on va réduire la précision des amplitudes à 12 bits et utiliser les 20 bits restants pour coder l'indice du trajet (cf. figure 2.14). De cette manière, un pixel contiendra bien l'amplitude maximum avec en plus l'information d'indice. En revanche, cette méthode pose

plusieurs problèmes :

- L'amplitude dispose d'un codage restreint : cela n'est pas un problème étant donné que les vues sont visualisées en niveaux de gris (ou en utilisant une palette de couleur spécifique) codée sur 256 valeurs.
- Le nombre de trajets va être limité à 2^{20} soit environ un million. Une grande majorité des jeux de données ayant un nombre de trajets inférieur, ce choix ne devrait pas être trop limitant.
- Pour un même trajet, il est possible d'avoir deux amplitudes identiques provenant de trajets différents. Dans ce cas, l'indice ne remontera que le trajet d'indice maximum. Une fois de plus, ce ne devrait pas être problématique dans la mesure où un trajet n'a pas plus d'importance qu'un autre.
- La précision réduite au niveau des amplitudes peut modifier le résultat de l'algorithme en cas d'amplitude identique au même pixel. Ce problème peut-être gênant car on ne sélectionne plus l'amplitude maximum de l'acquisition. Le passage de l'instruction atomique en 64 bits atténuera ce problème.

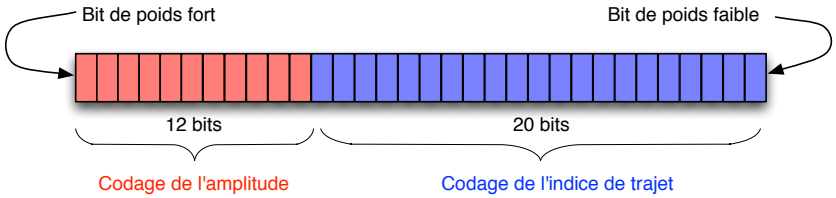


FIGURE 2.14 – Extension de l'utilisation de l'instruction atomique pour récupérer une seconde information.

Ces limitations devraient être assouplies une fois que les instructions atomiques double précision seront prises en charge par Nvidia dans leur implémentation OpenCL. Pour ce qui est des jeux de données de plus d'un million de trajets, il serait possible d'appliquer l'algorithmes en plusieurs passes contenant au maximum 2^{20} trajets.

Une fois cet indice récupéré, il est nécessaire d'effectuer un nouveau calcul pour récupérer le temps de vol et la profondeur de l'amplitude. En effet, il va être nécessaire de remonter au trajet utilisé pour récupérer les informations de temps de vol, profondeur et angle. Pour se faire, il est nécessaire de retrouver l'indice de l'amplitude utilisé dans le signal et la profondeur de projection sur le trajet.

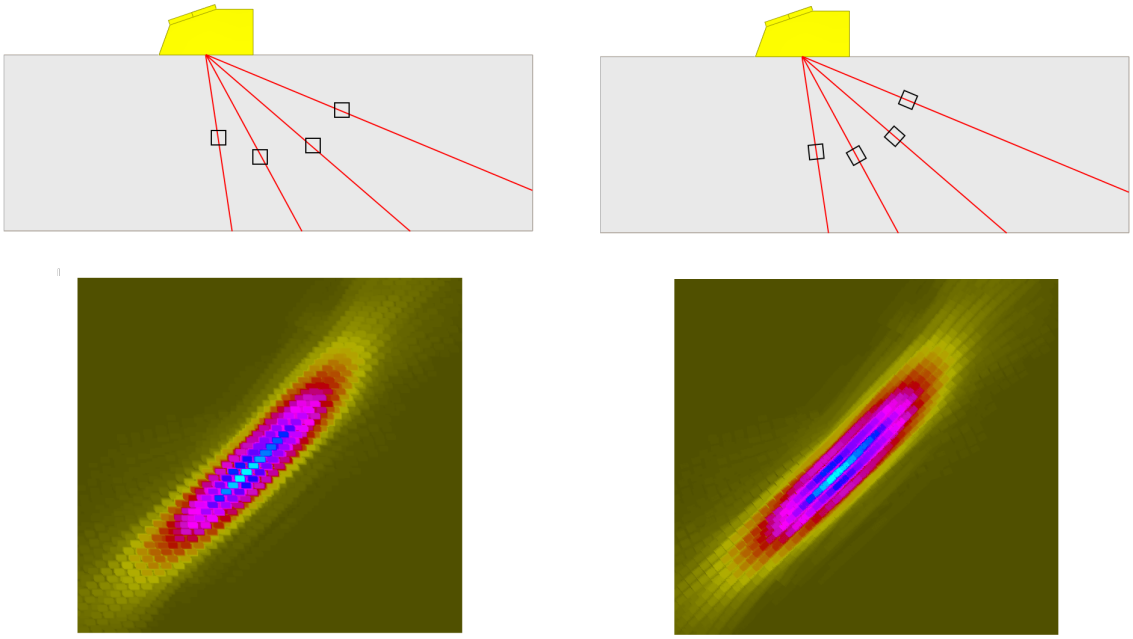
2.8.1.2 Extraction des informations supplémentaires

Un nouveau *kernel* a donc été développé pour extraire les informations de temps de vol et d'amplitude. Un simple parcours du signal qui correspond à l'indice n'est pas suffisant dans la mesure où plusieurs amplitudes d'un signal peuvent être égales. Il est donc nécessaire d'obtenir une information de localité.

Pour chaque pixel, on effectue une projection du segment de trajet dont provient l'amplitude, puis on parcourt cet intervalle à la recherche de cette amplitude. Si plusieurs valeurs

d'amplitudes sont égales à la valeur recherchée, nous prenons arbitrairement la première. La résolution des images utilisées étant suffisamment élevée, l'erreur est considérée comme suffisamment faible. Pour effectuer l'intersection entre la boite correspondent au pixel et le segment de trajet, on utilise un algorithme d'intersection optimisé [Williams et al., 2005]. Cette étape d'extraction est très peu coûteuse et ne dépasse par 1% du calcul de projection effectué au préalable. Une fois les informations récupérées, elles sont reportées dans l'interface de CIVA.

Le filtrage par dilatation n'est plus directement utilisable avec la notion d'angle. Le principe d'utilisation en post-traitement est, en revanche, conservé. Etant donné son faible coût, ce filtre n'a pas été reporté en OpenCL et c'est en Java côté *host* qu'il est effectué. Finalement, il va être possible d'obtenir une image plus lisse qu'avec le filtre initial. La figure2.15 présente la différence visuelle entre ces deux méthodes.



(a) Filtrage sans angle (b) Filtrage avec angle

FIGURE 2.15

2.8.2 Benchmark

Nous avons réalisé un ensemble de benchmarks pour valider les performances de l'intégration. Le tableau 2.12 présente les performances de reconstructions pour le jeu de données PMF (cf. section 2.3.3.1) pour des images de dimension 400×700. La machine utilisée est un GPP Intel Xeon E5540 cadencé à 2,53 GHz disposant de 4 coeurs (8 threads logiques avec *Hyperthreading*) et équipé d'un GPU Nvidia Tesla C2070. Nous avons fait le choix d'utiliser cette configuration car elle correspond aux machines déployées pour les développeurs et utilisateurs de CIVA en règle générale depuis 2010. Les GPP à 2 et 4 coeurs représentent aujourd'hui plus

Tableau 2.12 – Performances de l'implémentation OpenCL intégrée dans la plateforme CIVA

Temps (ms)	Vue de côté	Vue de dessus	Vue de face
CIVA 10 (Java monothread)	27861	23735	25032
CIVA 11 (Java monothread)	2880	2864	2871
OpenCL GPU	375	156	422
OpenCL GPP	841	784	806
Gain GPU/CIVA	$\times 74$	$\times 152$	$\times 59$
Gain OpenCL GPU/GPP	$\times 2$	$\times 5$	$\times 2$

de 90% du parc des machines utilisant CIVA.

Par rapport à CIVA 10, les gains sont évidemment très importants. Comme nous l'avons vu en section 2.2, ces gains proviennent du passage en post-traitement du filtrage d'image. Lorsque l'on optimise ce traitement, on obtient des temps de reconstruction légèrement inférieurs à 3 secondes. Les gains du GPU par rapport au GPP sont relativement attendus. Le quad-coeurs est évidemment moins véloce que le 12 coeurs évalué dans les benchmarks réalisés dans notre étude. Le gain sur GPU est donc réel sur ce type de machines, mais reste relativement faible (de l'ordre d'un facteur $\times 4$).

2.8.3 Conclusion

L'intégration de l'algorithme IVC a permis de soulever plusieurs problématiques discutées dans le chapitre 1 et plus particulièrement celle concernant l'évolution de codes optimisés (cf. section 1.4.4). En effet, la nécessité d'ajouter de l'information alors que les instructions atomiques des GPU ne permettent le traitement que d'une seule variable aurait pu être rédhibitoire. Un compromis a pu être trouvé en acceptant une perte de précision contrôlée, mais il est important de noter que cela aurait pu être différent.

Par rapport aux machines ciblées, principalement des GPP disposant de 4 coeurs, le GPU parvient à améliorer les performances d'un facteur allant de $\times 2$ à $\times 5$ sur le jeu de données évalué. Dans CIVA 10, le volume de données à traiter ainsi que le filtrage d'image associé rendait la reconstruction très peu confortable du fait du minimum de 20 secondes de calculs. Le prototype intégré permet, pour un résultat visuellement équivalent, de donner ce résultat en moins d'une seconde, que ce soit sur le GPP ou le GPU. Ces résultats sont donc très positifs.

2.9 Conclusion

Dans ce chapitre, nous avons présenté l'étude d'un premier algorithme de reconstruction basé sur la notion de trajet.

Nous avons pu voir qu'il était possible d'accélérer cet algorithme et nous avons apporté une solution efficace pour accélérer les calculs du traitement d'image en remplaçant un traitement à la volée très coûteux, par un traitement après projection permettant d'obtenir une image quasi-identique. Les facteurs d'accélération sont variables mais peuvent atteindre jusqu'à un facteur

$\times 40$.

Nous avons ensuite travaillé à sa parallélisation sur architectures GPP multicoeurs et GPU. Pour cela nous avons tout d'abord étudié l'approche de parallélisation, et ce, afin de déterminer la plus efficace. Nous avons montré que la parallélisation la plus efficace s'effectuerait sur les couples trajet/signal. Sur GPU, cette parallélisation implique d'utiliser de manière intensive les instructions atomiques du GPU. Ces instructions sont connues pour être peu performantes comparé à des accès classiques, mais l'évolution des GPU a permis des améliorations importantes que nous avons pu comparer en testant les architectures GPU GT200 et Fermi. De plus nous avons constaté que si Fermi a permis d'améliorer nettement les performances, les irrégularités dans les flux de calculs avaient toujours un impact important sur les performances de ces GPU.

Nous avons aussi montré que l'API OpenCL est tout à fait exploitable sur GPU et est à un niveau de performances proche à l'API propriétaire de Nvidia CUDA sur l'algorithme IVC. Malgré certaines fonctionnalités absentes de l'implémentation OpenCL de Nvidia, ainsi qu'un manque d'outils de développement important, nous avons porté le code depuis la source CUDA de manière relativement rapide.

Concernant les GPP multicoeurs, nous avons pu constater que, à la fois le Penryn 4 coeurs et le Nehalem 2×6 coeurs perrmettent d'obtenir des gains prévisibles. Le Nehalem nous a même permis d'obtenir un gain surlinéaire grâce à l'*Hyperthreading*, très bien exploité sur cette architecture. Nous nous sommes aussi intéressé à OpenCL, en utilisant l'implémentation OpenCL GPU, mais en utilisant les implémentations Intel et AMD d'OpenCL. Nous avons constaté un niveau de performances inférieur à ce que l'on pouvait obtenir avec OpenMP. Malgré cela les performances n'étaient que de l'ordre de 30% inférieures ce qui laisse penser qu'OpenCL pourrait tout à fait être utilisé à la place d'OpenMP dans un avenir assez proche. D'autant plus qu'OpenMP atteindra assez rapidement ses limites du fait de la structure de partage qui nécessite beaucoup de synchronisation sur le thread ordonnanceur.

L'étude de cet algorithme ne nous a pas permis de réellement départager les GPU des GPP. En comparant la plateforme GPP la plus performante du moment (2x6 coeurs Intel) aux GPU haut de gamme de Nvidia GeForce GTX 580 et Tesla C2070, nous avons constaté un niveau de performances relativement proche.

Enfin, nous avons réalisé l'intégration d'un prototype dans la plateforme logicielle CIVA à l'aide d'OpenCL. L'utilisation d'instructions atomiques aurait pu être limitante, mais nous avons apporté une solution satisfaisante permettant, au prix d'une perte de précision contrôlée, d'obtenir les informations nécessaires à la reconstruction. Les performances obtenues permettent à l'utilisateur de ces reconstructions d'obtenir des résultats en temps interactif avec des gains de performances allant d'un facteur $\times 50$ à un facteur $\times 150$ sur GPU par rapport à la version commerciale CIVA 10.

Nous allons maintenant passer à l'étude d'un second algorithme afin d'obtenir d'autres arguments nous permettant de départager ces deux architectures. Sachant que l'on cherche à orienter les futurs développements d'algorithmes de reconstruction ultrasons, il est important d'avoir une vision plus large. Pour cela, nous allons traiter un algorithme qui se base sur un autre type de simulation : la notion de temps de vol.

CHAPITRE 3

ETUDE DE LA PARALLÉLISATION DE L'ALGORITHME DE FOCALISATION EN TOUT POINT

Dans ce chapitre, nous allons étudier un algorithme de reconstruction d'ouverture synthétique : l'algorithme de Focalisation en Tout Point (FTP). Cet algorithme, basé sur le post-traitement de l'ensemble des données d'un contrôle ultrasons multi-éléments n'a pas toujours été très utilisé du fait de son coût en calcul très important.

Nous allons commencer par présenter la méthode FTP en détaillant particulièrement la première partie du calcul qui consiste à calculer les informations de simulation nécessaires à la reconstruction. Deux possibilités se présentent pour ce calcul : soit le traducteur est au contact de la pièce et il va être possible d'effectuer un calcul direct, soit le traducteur est en immersion et un calcul géométrique va être nécessaire pour obtenir les temps de vol. Nous allons plus particulièrement nous intéresser au calcul géométrique, calcul qui n'a pas encore fait l'objet d'accélération GPU ou multicoeurs dans la littérature.

L'analyse et les développements réalisés se basent sur la méthode FTP développée au CEA-LIST. Cette dernière utilise une architecture de calcul qui empêche de passer à l'échelle en terme de volumes de données du fait du pré-calcul des temps de vol. Pour cette raison, nous proposons deux méthodes de calcul permettant d'effectuer ce calcul à la volée. La première consiste en un calcul des temps de vol en tout point de la zone de reconstruction et la seconde utilise un principe d'interpolation se basant sur les propriétés physiques des ultrasons et réduisant significativement le coût de calcul.

Nous présentons les optimisations algorithmiques et architecturales nécessaires à l'implémentation optimale des deux nouvelles méthodes de calcul sur GPP et GPU. L'implémentation CUDA a été transcrite en OpenCL afin d'évaluer l'adaptabilité des GPP en terme de performance à utiliser un code GPU commun. Un ensemble de benchmarks couvrant un large ensemble de cas représentatifs pour le domaine d'applications a été réalisé et analysé afin de tirer le maximum d'informations des implémentations et architectures évaluées. Enfin, un proto-

type de la méthode la plus performante est implémentée dans CIVA et comparée en terme de performance à l'implémentation commerciale de CIVA.

3.1 Présentation de l'algorithme

Dans cette section, nous allons décrire les étapes et principes de l'algorithme de Focalisation en Tout Point. Nous commençons par le principe général, puis nous présentons la méthode de calcul de simulation permettant d'obtenir les temps de vol. Pour celle-ci, nous présentons les cas fonctionnels pris en compte et leurs spécificités.

3.1.1 Principe de la méthode de Focalisation en Tout Point

L'algorithme de Focalisation en Tout Point est un type d'algorithme de reconstruction ultrasons qui s'appuie sur une cartographie de temps de vol provenant de la simulation. Contrairement à l'algorithme présenté dans le chapitre précédent qui s'appuyait sur des trajets précalculés, on propose ici d'intégrer la phase de simulation à l'algorithme de reconstruction.

Ce type de méthode de reconstruction basée sur les temps de vol a été introduite dans le domaine du CND au travers de la méthode SAFT [Seydel, 1982]. Cette méthode est toujours très utilisée [Y. Tasinkevych, 2010] [Lévesque et al., 2002]. Du fait de l'évolution des capteurs ultrasons vers le multi-éléments, elle a évolué vers la méthode FTP [Holmes et al., 2004]. Celle-ci a déjà fait l'objet d'un portage sur GPU [Romero-Laorden et al., 2011b] mais uniquement dans le cas d'un capteur au contact de la surface pour lequel le calcul des temps de vol est trivial.

On associe à l'algorithme FTP un certain type d'acquisition appelé FMC (Full Matrix Capture) : considérons un traducteur ultrasonore multi-éléments de N_t éléments tel que présenté dans le chapitre 1 pour lequel chaque élément est utilisé successivement comme émetteur pendant que tous les autres sont utilisés comme récepteur. Les données acquises par ce type de contrôle sont mises en forme telle une matrice de signaux de taille $N_t \times N_t$. Cette matrice est nommée *Full Matrix Capture* ou FMC.

FTP est une technique qui permet d'exploiter les données de la FMC pour produire une image I, correspondant à une région de la pièce. Pour chaque point P de l'image le traducteur va être focalisé en émission et en réception. L'intensité de chaque pixel de l'image est donné par l'équation suivante :

$$I(P) = \sum_{i,j=1}^{N_t} S_{ij}(T_{ip} + T_{jp}) \tag{3.1}$$

où S_{ij} est le signal temporel acquis, T_{ip} et T_{jp} sont les temps de vol relatifs au point P pour les émetteurs i et j respectivement. Le calcul appliqué pour FTP peut donc se résumer de la manière suivante :

1. discrétisation sous forme de grille de la région à reconstruire,
2. calcul de $I(P)$ donné par l'équation 3.1 pour chaque point de la grille P.

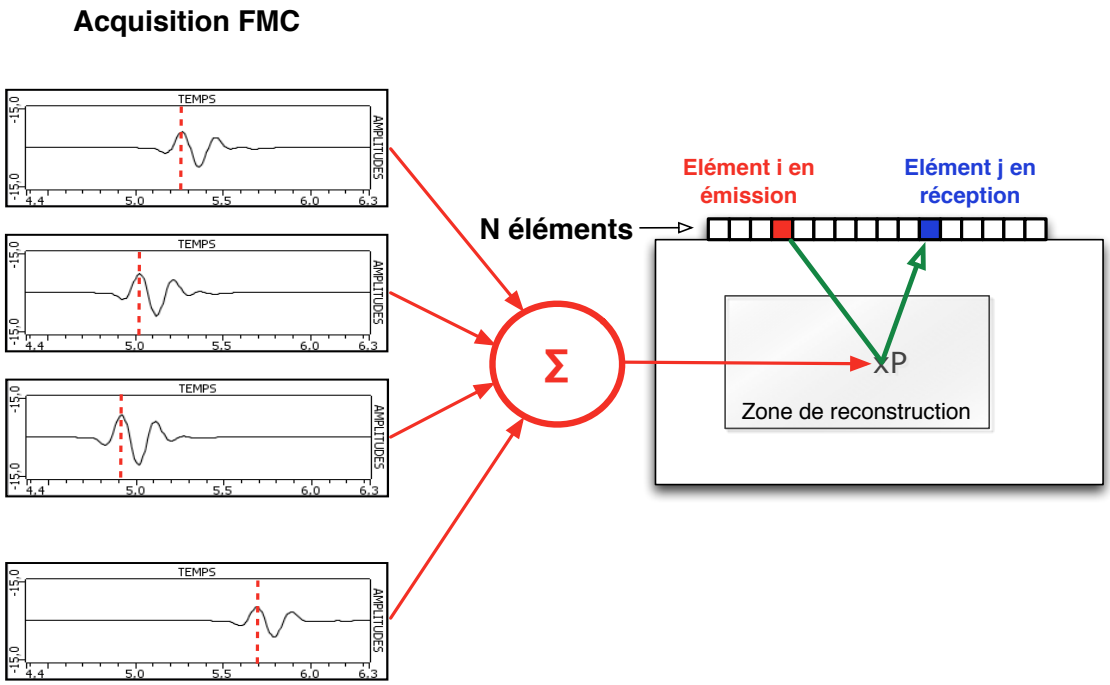


FIGURE 3.1 – L'algorithme FTP consiste en une focalisation par sommation cohérente de tous les signaux reçus $k_{ij}(t)$ pour tous les points P de la zone de reconstruction.

FTP peut être utilisé dans d'autres contextes d'acquisition selon le même principe d'association des données acquises à la grille de reconstruction en fonction des temps de vol simulés.

3.1.2 Calcul des temps de vol

Cette sous-section présente le calcul des temps de vol T_{ip}, $i = 1 \dots N$ pour des traducteurs au contact et en immersion pour des interfaces planes.

3.1.2.1 Traducteur au contact

Lorsqu'il est au contact, le traducteur est directement couplé à la pièce contrôlée en utilisant une couche très fine de couplant (sous forme de gel couplant ou d'une lame d'eau). La figure 3.2 présente ce type de contrôle. D'après [Holmes et al., 2005], le temps de propagation T_{ip} est déterminé en divisant la distance entre l'élément i et le point P par la vitesse du son c dans la pièce :

$$T_{ip} = \frac{\sqrt{(x_i - x)^2 + (z_i - z)^2}}{c} \tag{3.2}$$

où (x_i, z_i) est la position de l'élément i, et x et z sont les coordonnées du point P de l'image.

3.1.2.2 Traducteur sans contact pour une interface plane

En contrôle sans contact (immersion ou sabot, le traducteur est séparé de la pièce par un milieu couplant (eau ou sabot rigide en plexiglas). Dans ce mode d'inspection, l'onde va se

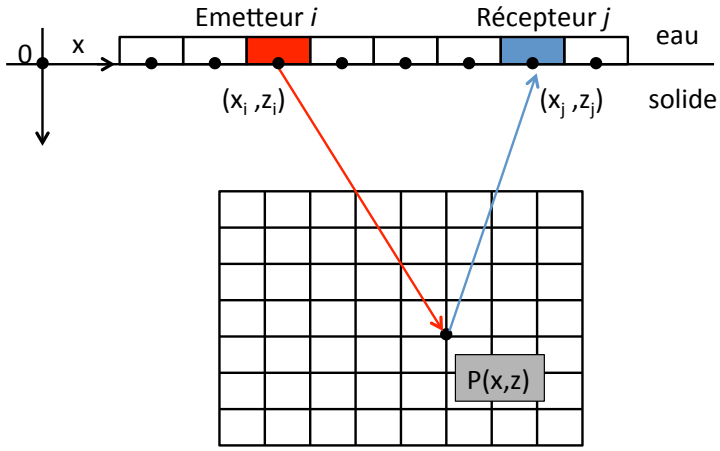


FIGURE 3.2 – Configuration de contrôle FTP en 2D pour un traducteur au contact.

propager dans deux milieux différents où les vitesses de propagation des ondes ultrasonores sont différentes. Cela implique un phénomène de réfraction à l'interface.

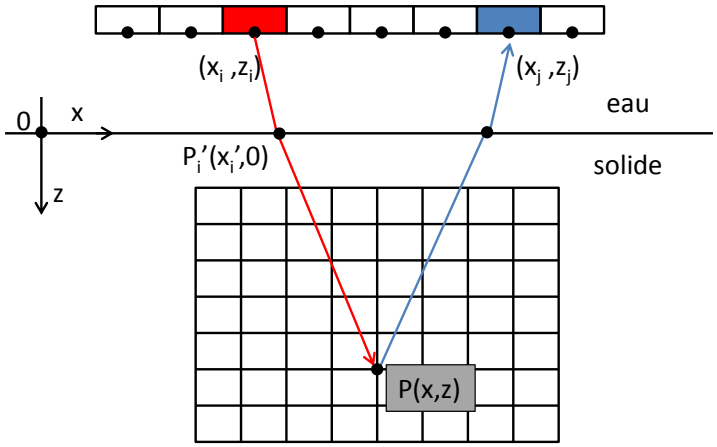


FIGURE 3.3 – Configuration de contrôle FTP en 2D pour un traducteur en immersion.

La figure 3.3 présente la configuration de contrôle pour ce cas où $P_i'(x_i', 0)$ est le point où l'onde est réfractée sur l'interface plane donnée par $z = 0$. De la même manière que pour le mode au contact, le temps de vol peut être exprimé de la façon suivante :

$$T_{ip} = \frac{\sqrt{(x_i - x_i')^2 + z_i^2}}{c_1} + \frac{\sqrt{(x_i' - x)^2 + z^2}}{c_2} \tag{3.3}$$

où c_1 et c_2 sont respectivement les vitesses de propagation dans le milieu couplant et dans la pièce, (x_i, z_i) est la position de l'élément i du traducteur, $(x'_i, 0)$ est la position du point de réfraction P' et (x, z) sont les coordonnées du point P de l'image.

La coordonnée x du point de réfraction P peut être déterminée en utilisant la loi de Snell-Descartes, ce qui mène, comme le détaille [Jobst and Connolly, 2010], à l'équation non linéaire suivante :

$$\frac{x_i - x'_i}{c_1\sqrt{(x_i - x'_i)^2 + z_i^2}} = \frac{x'_i - x}{c_2\sqrt{(x'_i - x)^2 + z^2}} \tag{3.4}$$

La solution de l'équation 3.4 peut être approximée en utilisant une méthode itérative de recherche de racines de polynômes. Le polynôme associé au cas présent est de degré 4. Avant d'aborder la méthode de résolution utilisée, nous allons présenter d'autres cas d'interfaces.

3.1.3 Interfaces complexes

Nous allons détailler les cas d'interfaces complexes. Tout d'abord, nous présentons les différents types d'interfaces qui peuvent être utilisées et ensuite puis nous aborderons le cas des pièces à interfaces multiples.

3.1.3.1 Types d'interfaces

Le cas sans contact direct avec la pièce présenté précédemment n'est valide que pour le cas d'une interface plane. En revanche, il est tout à fait possible de l'appliquer sur d'autres interfaces canoniques. Voici une liste de différentes interfaces possibles, ainsi que le degré du polynôme qui résulte de la mise en équation de la même manière que ci-dessus.

- **cylindre** : résulte en un polynôme dont le degré peut aller jusqu'à 10.
- **cône** : résulte en un polynôme dont le degré peut aller jusqu'à 14.
- **tore** : résulte en un polynôme dont le degré peut aller jusqu'à 16.

On peut tout de même noter quelques informations pour le cylindre. On peut considérer trois cas différents, selon l'alignement entre l'élément du traducteur et le point de l'image :

1. si l'élément du traducteur et le point de l'image sont tous deux sur un même plan formé par la génératrice du cylindre (cf. figure 3.4.a), il en résulte un polynôme de degré 4 tel que pour le cas plan,
2. si l'élément du traducteur et le point de l'image sont tous deux sur un même plan perpendiculaire à l'axe de la génératrice (cf. figure 3.4.b), alors il en résulte un polynôme de degré 6,
3. dans les autres cas, il résulte comme il est dit précédemment en un polynôme de degré 10.

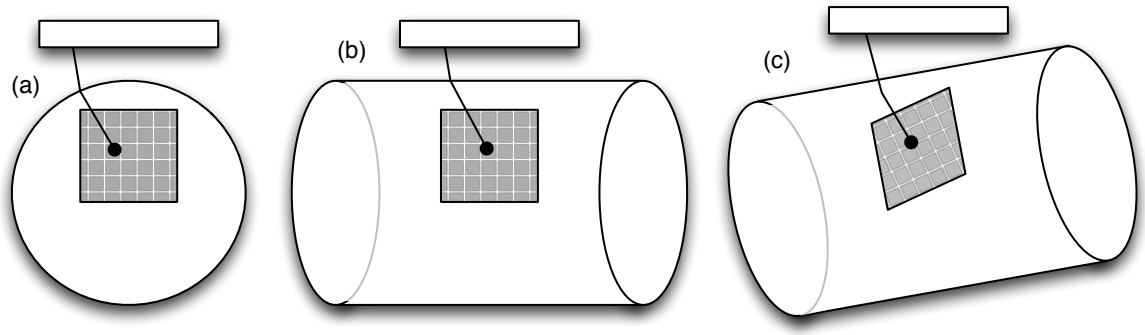


FIGURE 3.4 – Trois cas se présentant pour une interface cylindrique. Le polynôme résultant de la mise en équation passe du degré 4 à 6 puis 10, dans l'ordre des schémas (a), (b) et (c) respectivement.

Dans le cas de degré supérieur ou égal à 6, il est possible de trouver plusieurs solutions qui satisfassent la loi de Snell-Descartes. Dans ce cas, on ne conservera que le point d'impact permettant d'obtenir la distance la plus courte entre l'élément du capteur et le point de la pièce correspondant au pixel, en faisant l'hypothèse que ce sera le trajet prépondérant en énergie.

3.1.3.2 Géométries CAO 2D à interfaces multiples

Les types d'interfaces présentées précédemment sont utilisées pour la définition de modèles de pièces simples. Ces interfaces peuvent être assemblées pour créer des surfaces complexes. On va donc pouvoir associer plusieurs interfaces canoniques pour en créer d'autres plus complexes. Cela a un impact sur le calcul temps de vol. En effet, pour chaque point de calcul, on va itérer sur les interfaces et chercher l'ensemble des points qui valident Snell-Descartes. Comme pour les interfaces cylindriques, on n'utilise que le point d'impact permettant d'obtenir la distance la plus courte entre l'élément du traducteur et le point de pièce correspondant au pixel.

La figure 3.5 présente un exemple de points d'impacts multiples. Il y a deux interfaces planes et une cylindrique en vue de coupe. Si on applique Snell-Descartes entre le point $P0$ et $P2$ sur l'interface plane de gauche, on obtient un point d'impact $P1$. Lorsqu'on applique la même opération sur la surface cylindrique, on peut obtenir deux points, $P1'$ et $P2'$. Concernant l'interface plane de droite, aucun point d'impact ne satisfait l'équation de Snell-Descartes. Parmi les trois trajets trouvés, celui dont le point d'impact est $P1'$ est le plus court. On retient donc le temps de vol associé à ce trajet pour le point considéré.

Nous allons maintenant nous intéresser à la résolution d'équations polynomiales nécessaire au calcul des temps de vol.

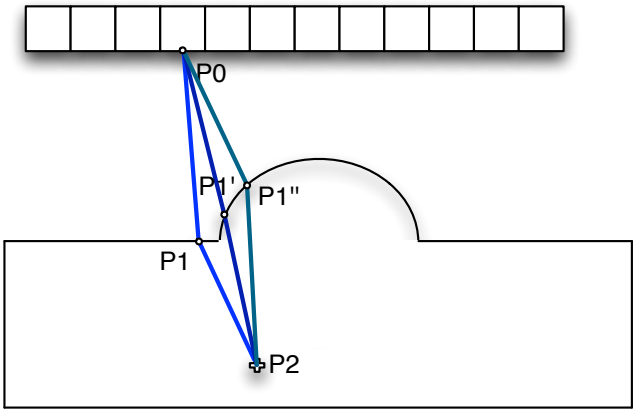


FIGURE 3.5 – Cas de CAO 2D multi-interfaces.

3.2 Recherche de zéros d'un polynôme

Comme nous l'avons dit dans la section précédente, il est nécessaire de trouver le point d'impact sur la surface de la pièce lors du calcul de temps de vol. Cette opération nécessite de déterminer les racines du polynôme associées au problème. Pour cela, on va utiliser une méthode itérative de recherche d'un zéro d'une fonction polynomiale qui va nous permettre de déterminer une valeur approchée de ces racines.

3.2.1 Choix de la méthode de résolution d'équations polynomiales

Il existe un nombre important de méthodes de recherches de racines d'un polynôme. Dans notre cas, nous nous intéressons à la résolution de polynômes de coefficients réels dont le degré est inférieur ou égal à 16. Nous ignorons les racines complexes, qui sont, du fait des coefficients réels du polynôme, des complexes conjugués, car nous recherchons une solution réelle. Nous cherchons donc une méthode appropriée à ce type de résolution.

Nous nous sommes intéressés à l'article de [Mekwi, 2001] qui reprend les différentes méthodes de résolution présentes dans la littérature et les compare en terme de complexité et de cas d'utilisation. Voici quelques rappels sur les méthodes de recherches de zéros :

Dans le cas d'un polynôme à coefficients réels, une racine complexe est forcément double (conjuguée). A chaque racine trouvée, on effectue une factorisation du polynôme en injectant cette ou ces racines dans le polynôme. Cette étape peut être critique numériquement car l'ordre d'obtention des racines peut faire varier les coefficients au degré inférieur et donc entraîner des erreurs ou des différences de plus en plus importantes à chaque changement de degré.

Ces méthodes de résolution prennent le plus souvent une approximation en entrée. Cela peut être très utile lorsque l'on a déjà l'information de l'intervalle où se trouve la ou les racines qui nous intéressent. Lorsque ce n'est pas le cas, il est intéressant d'avoir une méthode robuste qui va converger dans tous les cas, même si l'approximation nécessite plus de calculs.

Voici une liste des différentes méthodes avec pour chacune les points importants à noter, tout en sachant qu'il existe deux grands types de méthodes que sont les méthodes itératives et les méthodes qui permettent de trouver toutes les racines en même temps :

- **Méthode de Bairstow** : méthode itérative uniquement valide pour des coefficients réels. Un facteur réel multiple des deux racines est recherché afin de réduire les racines doubles. L'avantage principal de cette méthode est de ne travailler qu'avec une arithmétique réelle. En revanche, la vitesse de convergence quadratique quelle que soit le type de racine ainsi que la probabilité de non convergence non-nulle en font une méthode peu utilisée aujourd'hui.
- **Méthode de Newton** : méthode itérative probablement la plus connue. Elle se base sur la dérivée première du polynôme. La convergence de l'algorithme n'est prouvée que localement et la vitesse de convergence est quadratique elle aussi uniquement localement (sinon vitesse linéaire).
- **Méthode de Jenkins-Traub** : méthode itérative en trois étapes qui détermine les racines une à une. Elle est utilisée par Mathematica avec la fonction *NSolve*. Comme pour Newton ou Bairstow, une fois une racine trouvée, une factorisation du polynôme est effectuée. Cette méthode à l'avantage de converger globalement et sa vitesse de convergence est légèrement plus rapide que la méthode de Newton. En revanche, le nombre d'opérations par itération est environ deux fois plus importante que pour cette dernière.
- **Méthode de Laguerre** : méthode itérative très connue se basant sur la dérivée seconde en plus de la dérivée première (comme pour la méthode de Newton). Il est admis que la méthode est convergente de manière cubique localement pour des racines simples. Dans le cas de racines multiples, la convergence est linéaire. Cette méthode est une méthode de résolution généraliste.
- **Valeurs propres de la matrice compagnon** : méthode très précise et robuste se basant sur l'algorithme de décomposition QR. Elle est utilisée par Matlab avec la fonction *roots* ou par Scilab. Etant donné sa complexité, supérieure aux méthodes spécialisées dans la recherche de zéros, elle est souvent réservée aux cas de polynômes de degrés élevés (degré supérieur à 100).

Si certaines bibliothèques choisissent plutôt une méthode qu'une autre, il existe des bibliothèques qui proposent plusieurs méthodes, telle que la librairie IMSL Numerical bibliothèques qui en implémente trois, avec différentes recommandations selon le type de recherche de zéro à effectuer.

La méthode de Laguerre est un bon compromis compte tenu de nos critères : polynômes à coefficients réels, degré inférieur ou égal à 16 et recherche des racines réelles uniquement. En effet, sa robustesse dans la convergence permet d'éviter certains cas particuliers et sa convergence rapide pour les racines doubles nous a amené à nous orienter vers cette méthode. Ces arguments ont aussi permis à [Kajiya, 1982] de choisir Laguerre comme méthode de résolution pour des polynômes de degré 18. La méthode de Laguerre nous a semblé être tout à fait pertinente pour être portée sur nos architectures parallèles du fait de la quantité de calculs importante. Nous avons aussi gardé la possibilité d'utiliser une autre méthode telle que Jenkins-Traub, qui pourrait aussi tout à fait convenir pour notre type d'application.

3.2.2 Principe de la méthode itérative de Laguerre

L'algorithme 2 présente le pseudo-code de la méthode itérative permettant de trouver toutes les racines réelles d'un polynôme. La recherche de zéros de Laguerre est basée sur les formulations mathématiques issues de [Laguerre, 1879]. Selon le type de racine trouvée par la méthode de Laguerre, une factorisation correspondante est appliquée jusqu'à arriver au degré trois ou deux. Les coefficients du polynôme étant réels, une racine complexe implique que son conjugué est aussi racine ; il est donc possible d'effectuer une double factorisation.

Algorithme 2: Pseudo-code de la méthode itérative de recherche de racine basée sur la méthode de recherche de zéro de Laguerre.

sortie : Tableau R des racines réelles du polynôme
entrée : Degré d du polynôme
entrée : Polynôme $poly$

```
1  init ← 0
2  while d > 3 do
3      MiseEchelleCoefficients(poly, d)
4      racine ← Laguerre(init, poly, d)
5      if estComplexe(racine) then
6          FactorisationComplexe(poly, racine, d)
7          d ← d − 2
8      else
9          FactorisationReelle(poly, racine, d)
10         d ← d − 1
11         R ← racine
12 if d = 3 then
13     R ← ResolutionAnalytiqueFerrari(poly, d)
14 else
15     R ← ResolutionQuadratique(poly, d)
```

Pour les polynômes de degré inférieur ou égal à 3, nous utilisons les formules analytiques de Ferrari (publiées par Cardano dans son livre *Ars Magna*). Pour le degré 4, le calcul est plus complexe et il devient plus intéressant d'utiliser la méthode de Laguerre à la place des formules analytiques. Par rapport à la méthode de Laguerre proposée dans Numerical Recipes [Press et al., 2002] sur laquelle nous avons basé notre implémentation, nous utilisons une mise à l'échelle des coefficients permettant de minimiser les erreurs d'arrondis telles que dans l'implémentation de la méthode de Jenkins-Traub, *rpoly* [Jenkins, 1975]. Le critère de sortie de Laguerre utilisé est celui qui a été proposé par Adams [Adams, 1967] et permet un compromis entre l'utilisation maximale de la précision de la machine et le danger d'itérer alors qu'on a déjà dépassé l'erreur d'arrondi. Il est utilisé par Wolfram Mathematica [Weissten, 2012] et la bibliothèque NAG (*Numerical Algorithms Group*).

La robustesse et la justesse de la méthode de Laguerre sont prouvées en double précision. En revanche, les GPU sont aujourd'hui encore limités en terme de puissance de calcul double précision. Nous nous sommes donc posé la question de la robustesse de cette méthode pour une implémentation en simple précision (F32) sur les résultats et les performances de l'algorithme FTP.

Avant d'effectuer l'analyse en simple précision - que nous ferons avant de débuter les benchmarks, nous allons commencer par nous intéresser aux implémentations de l'algorithme.

3.3 Analyse algorithmique de FTP

Dans cette section, nous allons commencer par présenter un pseudo-code de l'algorithme de manière naïve. Nous présenterons ensuite une transformation algorithmique qui va nous permettre d'introduire deux algorithmes qui nous serviront de base pour les implémentations GPU et GPP. Le premier effectue un calcul exhaustif des temps de vol tandis que le second part d'une hypothèse physique permettant de réduire la quantité de calculs grâce à l'utilisation de calculs par interpolation.

3.3.1 Implémentation de référence et choix d'ordonnancement des calculs

L'implémentation de référence est actuellement intégrée dans CIVA. Nous sommes donc partis des connaissances déjà existantes sur le sujet pour proposer une alternative en vue d'une parallélisation GPP multicoeurs et GPU.

L'implémentation CIVA réalise un calcul en deux temps. Dans un premier temps, les temps de vol sont calculés sur un maillage plus lâche que la résolution pixel de la zone de reconstruction et les temps de vol pré-calculés et stockés en mémoire (nous allons aborder plus en détail l'interpolation dans la sous-section 3.3.4). Dans un second temps, ces temps de vol sont utilisés pour extraire les amplitudes des signaux et les sommer en chaque point de la zone, pour les différents couples émetteur-récepteur. Cet algorithme en deux passes est optimal pour le calcul des temps de vol. Par contre, le stockage en mémoire des temps de vol devient très vite limitant. Pour s'en affranchir, nous proposons d'effectuer un calcul à la volée.

Plusieurs possibilités se présentent quant à l'ordonnancement des boucles de calculs. En effet, comme l'équation 3.1 le montre, une somme d'opérations est effectuée pour chaque pixel traité. La boucle sur les pixels de l'image est implicite. Pour un émetteur e et un récepteur r et pour tout pixel p de l'image I, nous avons naturellement l'ordonnancement du nid de boucles suivant : p, e, r. L'algorithme 3 présente le pseudo-code de manière synthétique et naïve issu de l'ordre de boucle précédent. Nous pouvons constater que le calcul des temps de vol est systématique, pour tout pixel, et pour tout couple émetteur-récepteur. La complexité en terme de calculs de temps de vol est ici en $O(2N_t^2)$. Il est possible de sortir le calcul de T_{ip} et de le placer au niveau de la boucle sur i (algorithme 3 ligne 5) ce qui permet de réduire la complexité à $O(N_t^2)$.

Algorithme 3: Méthode FTP basée sur l'équation 3.1. Version naïve sans optimisation.

sortie : image I de dimensions $K \times L$
entrée : nombre d'émetteur-récepteur N_t du traducteur ultrasons
entrée : ensemble S des N_t^2 signaux
entrée : vitesse de propagation c des ultrasons dans le milieu contrôlé

```
   // Boucles sur les pixels de l'image
1  for i ← 1 to K do
2      for j ← 1 to L do
3          P ← P(i, j)
4          I(P) ← 0
           // Boucles sur les N_t^2 couples émetteur-récepteur
5          for e ← 1 to N_t do
6              for r ← 1 to N_t do
7                  T_eP ← CalculTempsVol(e, P)
8                  T_rP ← CalculTempsVol(r, P)
9                  I(P) ← I(P) + S_e,r(T_eP + T_rP)
```

Cet ordre de boucle est le plus avantageux dans la mesure où il va nous permettre d'éviter des calculs redondants comme nous allons le voir dans la sous-section suivante.

3.3.2 Egalité aller-retour des temps de vol

Comme nous l'avons vu, le calcul des temps de vol est coûteux pour les cas sans contact. Nous chercherons donc à minimiser la quantité de calculs nécessaires à la méthode FTP. Dans les cas qui nous intéressent, le mode de transmission des ondes ultrasonores va être identique à l'aller comme au retour. Le temps de vol entre un élément du traducteur ultrasons et un point de l'image va donc être identique que l'élément soit en émission ou en réception. Par conséquent, le temps de vol entre un élément i en émission et un point p va être identique au temps de vol entre ce même point p et le même élément en réception. Sur la figure 3.6), on a donc les égalités suivantes :

- $t_{a0} = t_{r1}$
- $t_{a1} = t_{r0}$
- $t_{a0} + t_{a1} = t_{r0} + t_{r1}$

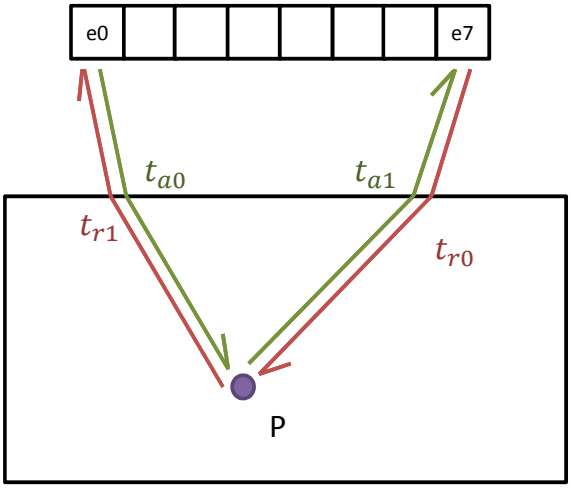


FIGURE 3.6 – Illustration de l'égalité de Helmoltz qui stipule que le signal reçu est le même lorsque des postes émetteurs et récepteur sont inversés ($t_{a0} = t_{r1}$).

Cette égalité vient du principe de Helmholtz qui stipule que le signal reçu est le même lors que les postes émetteur et récepteur sont inversés. Il a déjà été réutilisé dans le cadre d'implémentations optimisées de FTP au contact pour limiter la quantité de signaux à acquérir ou à stocker [Holmes et al., 2008] [Sutcliffe et al., 2012].

Pour notre application, nous allons uniquement nous intéresser à l'égalité au niveau temps de vol. Pour résumer, pour un émetteur e, un récepteur r et un point p de l'image, nous avons l'égalité suivante :

$$e = r \Rightarrow T_{ep} = T_{rp} \tag{3.5}$$

Cette égalité nous permet de réduire le nombre de calculs de temps de vol nécessaire puisqu'avec les temps de vol en émission T_{ip}, il va être possible de calculer tous les temps de vol de chaque couple émetteur-récepteur passant par un point p.

3.3.3 Algorithme par calcul exhaustif des temps de vol

L'algorithme 4 présente le calcul exhaustif des temps de vol. Le principe de réciprocité permet de réduire la complexité en terme de calcul des temps de vol à $O(N_t)$.

Nous obtenons donc ici un algorithme bien plus performant que la formulation initiale. Néanmoins, dans cette version, l'algorithme n'utilise pas de maille d'interpolation sur les temps de vols. Cette optimisation supplémentaire est possible en fonction de la résolution de la grille de reconstruction vis-à-vis de la longueur d'onde du signal. Nous allons donc maintenant voir de quelle manière l'algorithme évolue lorsqu'on ajoute cette maille d'interpolation.

Algorithme 4: Méthode FTP avec utilisation de l'égalité des temps de vol entre émetteur et récepteur.

sortie : image I de dimensions $H \times W$
entrée : nombre d'émetteur-récepteur N_t du traducteur ultrasons
entrée : ensemble S des N_t^2 signaux
entrée : vitesse de propagation c des ultrasons dans le milieu contrôlé

```
// Boucles sur les pixels de l'image
for i ← 1 to H do
    for j ← 1 to W do
        P ← P(i, j)
        I(P) ← 0
        // Boucle sur les éléments du traducteur ultrasons
        for t ← 1 to N_t do
            T(t) ← CalculTempsVol(t, P)
        // Boucles sur les N_t^2 couples émetteur-récepteur
        for e ← 1 to N_t do
            for r ← 1 to N_t do
                I(P) ← I(P) + S_{e,r}(T(e) + T(r))
```

3.3.4 Algorithme par interpolation des temps de vol

Le CEA propose d'utiliser une grille d'interpolation afin d'éviter un calcul des temps de vol dépendant de la résolution pixels souhaitée par l'utilisateur. Cette grille permet de différencier la résolution physique liée à la fréquence des ondes ultrasons émises et aux vitesses dans les différents milieux qu'elles traversent, et la résolution pixel demandée par l'utilisateur. La figure 3.7 présente la différence entre les deux niveaux de maille. L'objectif de cette maille est d'éviter des calculs précis de valeurs dont l'estimation par interpolation suffirait.

Pour obtenir une précision suffisante, on considère que l'interpolation doit être effectuée sur une maille de la taille de la longueur d'onde des ondes ultrasons dans le milieu contrôlé. Par exemple, pour un traducteur émettant des ondes à 2 MHz, et une pièce en acier où les ondes sont propagées à 6000 $m.s^{-1}$, la longueur d'ondes λ est de 3 mm. Pour ce type de contrôle ultrasonore, les points de la grille d'interpolation sont donc espacés de cette distance. L'utilisateur de l'algorithme va dimensionner sa zone de reconstruction en fonction de la longueur d'onde dans le milieu contrôlé. La résolution adaptée se trouve autour de $\lambda/10$. Le nombre de pixels par tuile est donc variable selon les configurations de reconstruction mais sera de l'ordre de 100 à 250 pixels par tuile.

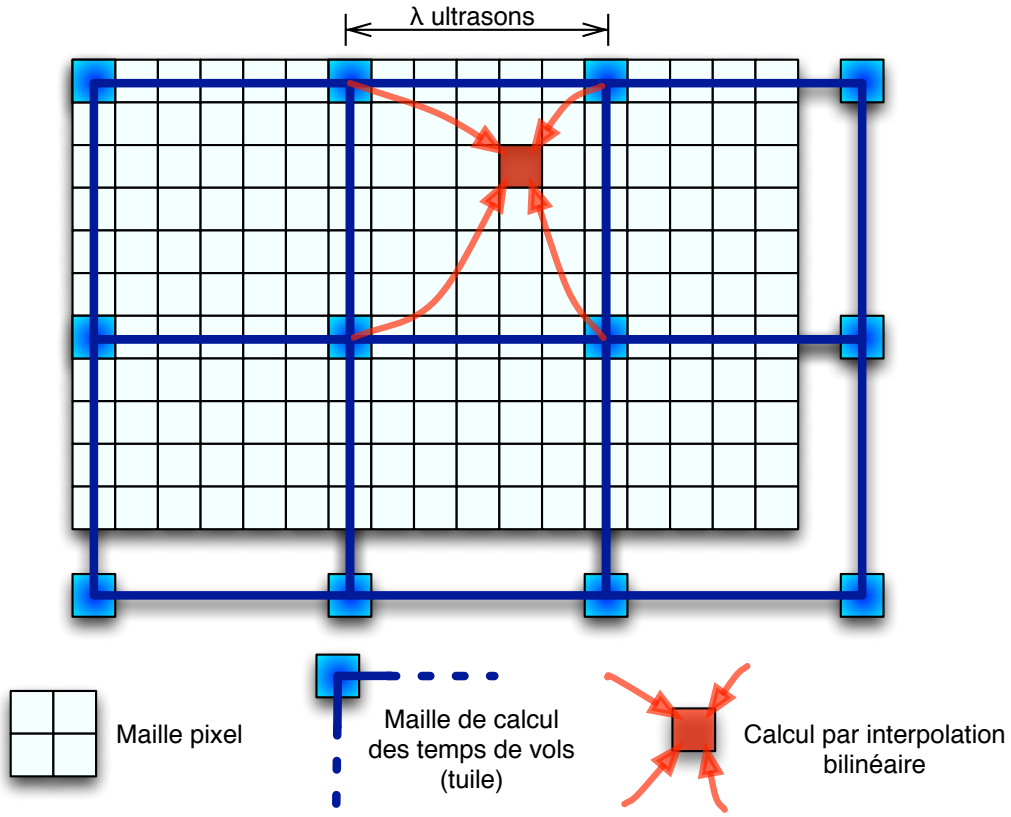


FIGURE 3.7 – Schéma présentant les deux niveaux de maillages utilisés pour les différentes approches de calcul utilisées pour le calcul de FTP. On distingue une maille intermédiaire lâche qui permet de réduire la quantité de calcul en minimisant la perte de précision (du fait de l'utilisation de la longueur d'onde des ultrasons dans la pièce). Les temps de vol sont calculés par interpolation bilinéaire sur les coins de chaque tuile.

L'algorithme 5 présente le pseudo-code de l'implémentation par interpolation sur les temps de vol. L'algorithme ne diffère pas en terme d'étapes par rapport à l'algorithme 4. La première étape est toujours dédiée au calcul des temps de vol et la seconde à l'extraction des amplitudes et sommation. La différence est que l'on utilise une grille intermédiaire, plus lâche que la grille pixel, pour le calcul des temps de vol. L'étape 1 est donc utilisée pour calculer les temps de vol aux quatre coins d'une tuile de la maille de calcul. L'étape 2 se charge de charger les temps de vol gardés en mémoire dans les tableaux temporaires T_{Px} et d'interpoler ces quatre valeurs selon la position de pixel au sein de cette tuile (c.f. figure 3.7). Une fois les temps de vol en émission et en réception calculés, l'étape d'extraction d'amplitude et sommation est la même que pour les autres. Il est à noter qu'une implémentation 3D est possible sur le même modèle. Nous en discuterons plus tard lors de l'intégration.

Algorithme 5: Méthode FTP avec optimisation du calcul des temps de vol et utilisation d'un calcul par interpolation.

sortie : image I décomposée en une grille de $X \times Y$ tuiles de dimensions $K \times L$ pixels.
entrée : nombre d'émetteur-récepteur N_t du traducteur ultrasons
entrée : ensemble S des N_t^2 signaux
entrée : vitesse de propagation c des ultrasons dans le milieu contrôlé

```
// Boucles sur points de grilles de tuiles de la maille physique.
for x ← 1 to X − 1 do
    for y ← 1 to Y − 1 do
        // Coins de la tuile
        P0 ← P(x + 0, y + 0)
        P1 ← P(x + 1, y + 0)
        P2 ← P(x + 0, y + 1)
        P3 ← P(x + 1, y + 1)
        // Boucle sur les éléments du traducteur ultrasons
        for t ← 1 to N_t do
            T_P0(t) ← distance(t, P0)/c
            T_P1(t) ← distance(t, P1)/c
            T_P2(t) ← distance(t, P2)/c
            T_P3(t) ← distance(t, P3)/c
        // Boucle sur les pixels contenus par la tuile
        for k ← 1 to K_t do
            for l ← 1 to L_t do
                I(P_{k,l}) ← 0
                // Boucles sur les N_t^2 couples émetteur-récepteur
                for e ← 1 to N_t do
                    for r ← 1 to N_t do
                        T_e ← InterpolationBilinaire(P_{k,l}, P0, P1, P2, P3,
                                T_P0(e), T_P1(e), T_P2(e), T_P3(e))
                        T_r ← InterpolationBilinaire(P_{k,l}, P0, P1, P2, P3,
                                T_P0(r), T_P1(r), T_P2(r), T_P3(r))
                        I(P(k, l)) ← I(P(k, l)) + S_{e,r}(T_e + T_r)
```

D'autres pistes d'optimisations sont à l'étude au CEA, comme par exemple l'interpolation au niveau de l'image finale. Un maillage lâche tel que celui utilisé pour les temps de vol, mais dimensionné de manière différente, est utilisé pour effectuer la reconstruction. Ensuite, les amplitudes sont interpolées. Nous ne nous sommes pas intéressés à cette méthode car elle demandait une validation physique supplémentaire par rapport à l'existant.

Nous allons maintenant nous intéresser aux aspects de complexité algorithmique des différentes méthodes de calcul.

3.3.5 Complexité algorithmique

Cette section propose une analyse de la complexité algorithmique des différentes approches présentées ci-dessus. Nous nous plaçons dans le cas de configurations avec un traducteur multi-éléments de N_t émetteur-récepteurs ainsi qu'une image de N_p pixels. Ces deux paramètres font varier les boucles principales des algorithmes présentés dans la section précédente. On va donc pouvoir exprimer la complexité des différentes approches en utilisant ces termes.

Le tableau 3.1 présente les complexités en terme d'accès mémoire et calcul des temps de vol pour les différents algorithmes. Il est à noter que si l'implémentation naïve est présente, nous ne l'utiliserons pas dans la partie implémentation dans la mesure où sa complexité en terme de calculs et d'accès mémoire est trop élevée : $O(N_t^2 \times N_p)$ contre $O(N_t \times N_p)$ pour l'approche avec optimisation et sans interpolation.

Tableau 3.1 – Tableaux présentant les complexité en terme d'accès mémoire et en terme de calcul des temps de vol pour les différentes approches de calcul de la méthode FTP.

Complexité algorithmique	Accès mémoire	Calcul des temps de vol
Systématique sans optimisation	$O(N_t^2 \times N_p)$	$O(N_t^2 \times N_p)$
Systématique avec optimisation	$O(N_t^2 \times N_p)$	$O(N_t \times N_p)$
Interpolation avec optimisation	$O(N_t^2 \times N_p)$	$O(N_t \times N_{pi})$

La complexité de calcul de l'implémentation avec interpolation ne dépend plus du nombre de pixel N_p mais bien du nombre de points de la maille de calcul utilisée pour l'interpolation noté ici N_{pi}. Comme nous l'avons dit dans la section précédente, ce nombre est variable mais est de l'ordre de 10^2, donc : $N_p \simeq 10^2 \times N_{pi}$. En termes d'accès mémoire, les deux algorithmes optimisés ont la même complexité dans la mesure où le nombre d'accès aux signaux est le même.

3.3.6 Description des données

Comme pour l'algorithme étudié dans le chapitre précédent, les données sont soient issues d'une acquisition, soit d'une simulation de CIVA. Ces données sont stockées dans un fichier sur disque dur. Nous allons présenter les différentes informations nécessaires à la reconstruction. Nous allons effectuer un chargement des signaux en mémoire RAM.

3.3.6.1 Signaux

Les signaux sont stockés de manière contigüe en mémoire. Dans notre cas, nous travaillons sur des signaux de taille identiques pour un contrôle donné. A la différence de l'algorithme précédent, nous travaillons ici avec un traducteur multi-éléments. Les signaux sont stockés dans l'ordre des couples émetteur-récepteurs, sachant que l'on fixe d'abord un émetteur puis que l'on boucle sur les récepteurs. On a donc une structure telle que présenté sur la figure 3.8. Chaque signal est décomposé en échantillons temporels.

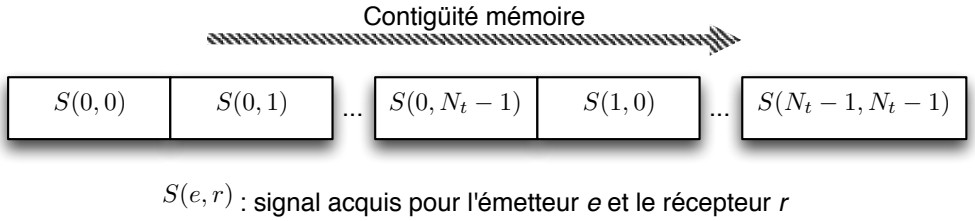


FIGURE 3.8 – Schéma présentant le format mémoire adopté pour stocker les signaux issus d'une acquisition ou simulation ultrasons avec un traducteur multi-élément.

3.3.6.2 Variables associées au contrôle

Nous avons différents types d'informations à passer à notre code de calcul. Ce sont les variables dépendant de la configuration du contrôle :

- **Les surfaces** : selon les surfaces sur lesquelles on effectue le calcul, les définitions varient. En effet, pour une surface plane, 4 points en 3D suffisent, alors que pour un cylindre on a besoin d'un point, d'un vecteur pour la génératrice, d'un vecteur pour la direction de l'ouverture, de la dimension de l'ouverture. Toutes ces informations sont stockées avec des structures afin de pouvoir les utiliser, à la fois dans les implémentations C++, ainsi que dans les implémentations CUDA ou OpenCL,
- **la pièce** : matériau homogène ce qui implique une vitesse des ondes ultrasonores fixe pour une pièce donnée.,
- **Ll capteur** : représenté par une position, une taille par élément, une taille inter-éléments, ainsi que la fréquence centrale d'émission,
- **l'image** : représentée par une position dans la pièce, d'une dimension en millimètres et d'une résolution en pixels.

3.3.6.3 Dimensionnement des configurations

Comme nous le verrons par la suite, nous nous intéressons plus particulièrement à des traducteurs de 128 éléments bien qu'il soit nécessaire de gérer un nombre d'éléments variables entre 16 et 256. Quant aux images, elles sont de résolutions variables pouvant être comprises entre 10 000 et 100 000 pixels.

Etant donné que l'approche de calcul par interpolation nous rend moins dépendant de la résolution pixel du fait de la réduction de la quantité de calculs, il faut tout de même noter que nous travaillerons avec un ratio entre le nombre de points de calculs en interpolation et le nombre de pixels compris entre 0,1 et 0,2. Cela représente donc en deux dimensions des tuiles comprenant entre 100 et 400 pixels. Cet ordre de grandeur est relativement élevé et devrait permettre au calcul par interpolation d'être très efficace.

3.3.7 Approches de parallélisation

Différentes possibilités s'offrent à nous quant à la parallélisation des algorithmes 4 et 5. Néanmoins, comme nous l'avons vu, il va être intéressant de profiter de la réutilisabilité des temps de vol afin de réduire la quantité de calculs (section 3.3.2). L'IEF a d'ailleurs confirmé ce besoin dans le cadre d'une étude du code. Etant donné que la réduction de complexité se situe pour un point de reconstruction, il est donc nécessaire d'effectuer un découpage et donc une parallélisation sur les pixels de l'image.

La quantité de calcul par pixel devrait être largement assez élevée pour le GPP, en masquant les *overheads* d'ordonnancement ou de chargement des threads sans avoir à utiliser des techniques telles que du déroulage de boucle. Quant au GPU, le nombre de pixels de l'image devrait être suffisant à charger entièrement les GPU. Dans le cas du GPU, les implémentations réalisés sont très dépendantes du matériel du fait de l'association entre les données et la taille de la mémoire partagée. Par conséquent, nous donnerons tous les détails sur les implémentations dans la partie benchmark dédiée au GPU. Quant au GPP, les implémentations sont aussi discutées dans la partie benchmark dédiée au GPP.

De manière synthétique, nous pouvons dire que la parallélisation de l'algorithme par calcul exhaustif va être effectuée sur les pixels, tandis que l'algorithme par calcul par interpolation le sera sur les tuiles. Nous allons maintenant passer à la présentation du benchmark.

3.4 Préparation du benchmark

Dans cette section, nous allons présenter les architectures et logiciels utilisés, puis nous présenterons les jeux de données qui vont être utilisés pour les différents tests et nous terminerons par une discussion sur la précision des résultats.

3.4.1 Architectures et logiciels

Même si les architectures étudiées sont en partie les mêmes que pour le chapitre précédent, nous avons effectué quelques modifications dans cette liste. Il ne nous a pas paru pertinent de tester à nouveau l'ancienne génération de GPU. Par conséquent, la Tesla C1060 ne sera pas évaluée pour ce benchmark, du fait de son retard en termes de performances comparé à la génération Fermi et aussi au fait que la nouvelle génération Nvidia arrive sur le marché actuellement via des cartes milieu de gamme. Nous avons aussi choisi de retirer le Xeon Penryn, du fait de son petit nombre de coeurs, et donc de la limite des informations que l'on pouvait obtenir. Nous avons aussi ajouté une carte GPU AMD Radeon HD6970 afin d'évaluer leur capacité sur un code développé initialement pour les GPU Nvidia. Le tableau 3.2 présente donc les architectures évaluées par le benchmark qui suit.

Tableau 3.2 – Tableaux présentant les architectures évaluées avec le benchmark FTP.

Architectures	# coeurs	Fréquence (GHz)	Bande passante (GB/s)	GFLOPs simple précision (scalar/SIMD)	GFLOPs double précision (scalar/SIMD)
Xeon X5690 (2)	12	3,47	64	80/320	80/160
GeForce GTX 580	512	1,54	192	1580	198
Tesla C2070	448	1,15	143	1030	515
Radeon HD 6970	1536	0,88	176	2700	683

Au niveau logiciel, voici les différentes versions de logiciels et de pilotes utilisées :
- CUDA 4.2 (compilateur basé sur la LLVM) et le driver 301.32,
- OpenCL Nvidia associé à CUDA 4.2,
- OpenCL Intel 1.5,
- OpenCL AMD 2.7 (implémentant la norme OpenCL 1.2) et le driver Catalyst 12.6,
- Compilateur C++ Visual Studio 2010 implémentant la norme OpenMP 2.5.

Les mesures de temps ont été effectuées de la manière présentée dans la section 1.4.7.

3.4.2 Présentation des jeux de données

Quatre jeux de données ont été sélectionnés pour réaliser les différents benchmarks. Parmi ces quatre, quelques variations seront utilisées mais nous les détaillerons durant les benchmarks eux mêmes.

Nous avons choisi un dimensionnement spécifique selon les différents fichiers. Le dimensionnement des images influe directement sur la parallélisation ainsi que sur les différences de performances entre la méthode de calcul sans interpolation et la méthode par interpolation. Nous avons cherché à utiliser des paramètres réalistes par rapport au besoin d'accélération de l'utilisateur.

Les quatre fichiers utilisent un traducteur multi-élément de 128 éléments en immersion. Cette valeur est relativement importante par rapport à l'ordre de grandeur utilisé actuellement dans les milieux industriels, mais est aussi le type de traducteur pour lequel les temps de calculs de reconstruction peuvent être un frein à leur utilisation. Voici une liste des quatre fichiers et les illustrations leur correspondant :

Plan - figure 3.9
- type de pièce : simple composée d'un plan unique,
- fréquence émetteur : 2 MHz,
- nombre d'échantillons par signal : 1031,
- dimension image physique (mm) : 40×40,
- dimension image visualisée (pixels) : 200×200,
- volume de données : 130 Mo.

Cyl - figure 3.10
- type de pièce : simple composée d'un cylindre unique (traducteur perpendiculaire à la génératrice par défaut),
- fréquence émetteur : 2 MHz,
- nombre d'échantillons par signal : 2007,
- dimension image physique (mm) : 40×40,
- dimension image visualisée (pixels) : 200×200,
- volume de données : 250 Mo.

PCP - figure 3.11
- type de pièce : multi-surfaces composée de 2 plans et 1 cylindre (traducteur perpendiculaire à la génératrice par défaut),

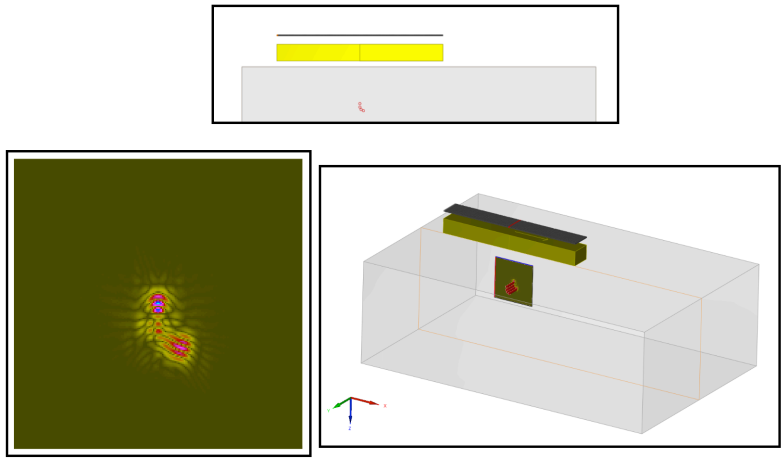


FIGURE 3.9 – Dans l'ordre de lecture, vue de coupe du contrôle, vue reconstruite par FTP et enfin vue replacée dans la scène 3D pour le jeu de données Plan.

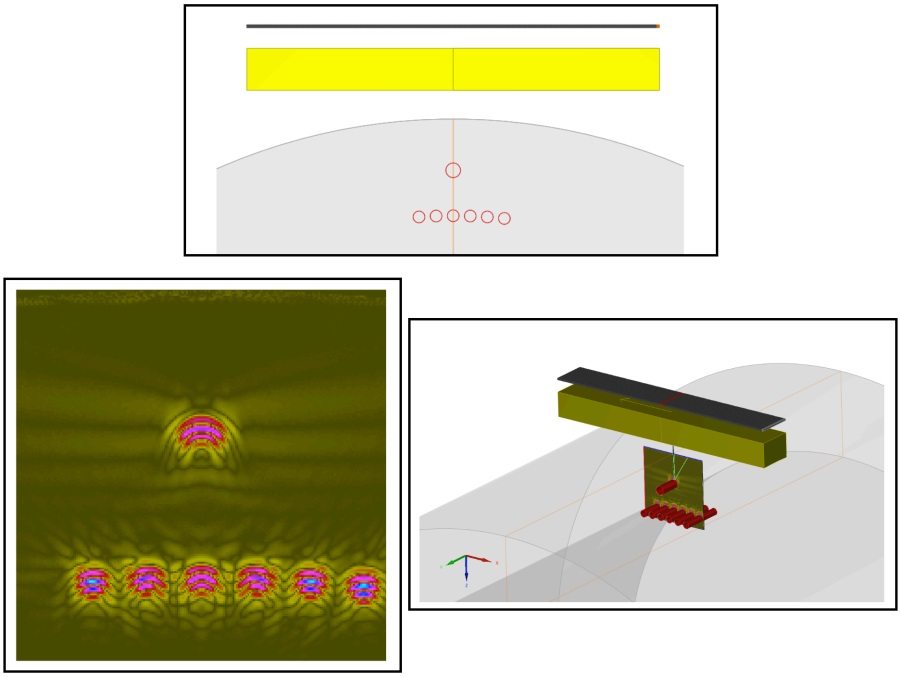


FIGURE 3.10 – Dans l'ordre de lecture, vue de coupe du contrôle, vue reconstruite par FTP et enfin vue replacée dans la scène 3D pour le jeu de données Cyl.

- fréquence émetteur : 2 MHz,
- nombre d'échantillons par signal : 1382,
- dimension image physique (mm) : 40×40,
- dimension image visualisée (pixels) : 200×200,
- volume de données : 170 Mo.

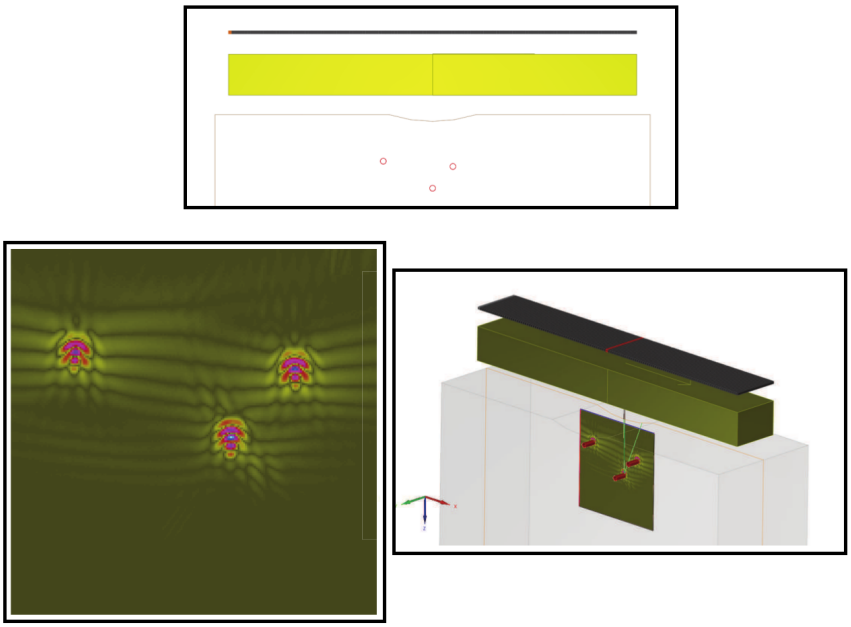


FIGURE 3.11 – Dans l'ordre de lecture, vue de coupe du contrôle, vue reconstruite par FTP et enfin vue replacée dans la scène 3D pour le jeu de données PCP.

P4CP - figure 3.12

- type de pièce : multi-surfaces composée de 2 plans et 4 cylindres (traducteur perpendiculaire à la génératrice des 4 cylindres par défaut),
- fréquence émetteur : 5 MHz,
- nombre d'échantillons par signal : 750,
- dimension image physique (mm) : 180×47,
- dimension image visualisée (pixels) : 900×235,
- volume de données : 100 Mo.

Les volumes de données sont relativement réduits car on travaille sur des simulations ou acquisitions qui ne sont faites qu'à une seule position mécanique. En pratique, l'utilisateur va boucler sur ces positions. En termes de calculs, cette étape est itérative et nous ne nous y sommes pas intéressés. La problématique principale de ces itérations est le volume de données à traiter qui va très vite dépasser la taille de la DRAM des GPU. Nous avons initialement laissé cette problématique de côté.

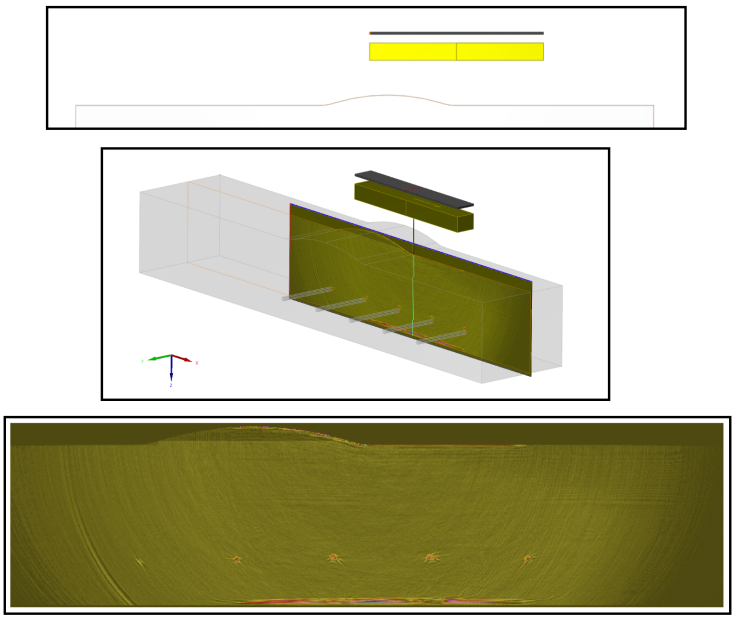


FIGURE 3.12 – Dans l'ordre de lecture, vue de coupe du contrôle, vue replacée dans la scène 3D et enfin vue reconstruite par FTP pour le jeu de données P4CP. La forme arrondie en surface est composée de quatre arcs de cercle différents.

3.4.3 Type de données et précision des résultats

Les GPU actuels n'étant pas complètement double précision, nous avons considéré la possibilité d'utiliser un type `float` (F32) à la place du type `double` (F64) utilisés dans le code d'origine. Nous nous sommes donc intéressés à la précision des résultats obtenus en reconstruction fonction de ces types. Nous avons abordé le problème par deux méthodes.

La première méthode a été effectuée en deux étapes. La première étape consiste à comparer la précision des racines utilisées par la reconstruction pour les résolutions polynomiales F32, F64 et *Solve* de Mathematica afin de déterminer une erreur maximale pour un grand ensemble de polynômes. La seconde consiste à obtenir une erreur sur l'accès aux échantillons de signaux fonction de cette erreur calculée précédemment. Les deux premières sous sections sont consacrées à cette méthode.

La seconde méthode s'est basée sur l'utilisation du logiciel CADNA, développé au LIP6 (Université Pierre et Marie Curi)e [Scott et al., 2007]. CADNA utilise une approche stochastique permettant d'évaluer les erreurs d'arrondi lors de l'exécution d'un code. Nous présenterons les concepts de la méthode ainsi que le logiciel CADNA, et la manière dont nous l'avons utilisé pour évaluer la propagation des erreurs d'arrondis dans la méthode de Laguerre.

3.4.3.1 Etude statistique de la méthode de recherche de zéros d'une fonction polynomiale

Nous cherchons à valider une méthode de résolution pour des polynômes de degré 16 au maximum (cas torique). Pour cela, nous avons entrepris de valider notre résolution sur des polynômes générés par CIVA pour ce cas torique. 100000 polynômes ont été générés, représentant un cas de reconstruction FTP pour un traducteur multi-éléments et une zone de reconstruction 3D, et ce, afin d'obtenir un maximum de cas différents.

Le tableau 3.3 présente des résultats comparatifs entre la résolution F64 et la résolution F32. Les résultats sont comparés à la résolution de Mathematica obtenus via la fonction *Solve*.

Tableau 3.3 – Comparaison F32/F64 pour la méthode de recherche de zéros d'un polynôme.

Résolution de 100k polynômes de degré 16	F64	F32
Nombre moyen d'itérations au total	5,6	4,7
Médiane nombre itérations au total	5,0	5,0
Ecart-type nombre itérations au total	1,1	1,0
Nombre moyen d'itérations pour la racine choisie	5,6	4,8
Médiane nombre itérations pour la racine choisie	6,0	5,0
Ecart-type nombre itérations pour la racine choisie	0,8	0,7
Nombre de polynômes dont le nombre de racines diffère de Mathematica (Solve)	0	100
Moyenne valeurs racines	$2{,}57 \cdot 10^{-15}$	$7{,}01 \cdot 10^{-7}$
Médiane valeurs racines	$2{,}1 \cdot 10^{-16}$	$1{,}04 \cdot 10^{-7}$
Erreur maximum (mm)	$1{,}28 \cdot 10^{-10}$	$3{,}38 \cdot 10^{-4}$

Sur les six premières lignes du tableau, nous remarquons que le nombre d'itérations en F32 est légèrement inférieur à l'implémentation F64. Sachant que l'on est en flottants, la plus petite valeur qui lorsque qu'on lui ajoute 1 donne une valeur différente est en F32 `FLT_EPSILON` et en F64 `DBL_EPSILON`. Etant donné que le critère de sortie de Laguerre est basé sur cette valeur (cf. section 3.2.2), il est normal de réaliser moins d'itérations pour atteindre la précision maximum.

Ensuite, on constate que le nombre de racines n'est pas le même entre Mathematica/F64 et F32. Cet écart s'explique par le critère d'Adams cité ci-dessus. La sortie du calcul de Laguerre est donc prématurée mais nécessaire pour éviter une erreur numérique plus importante. Pour ces polynômes, deux racines réelles sont trouvées à la place d'une racine complexe conjuguée. Ce type de débordement pourrait être problématique puisque les racines réelles trouvées s'ajoutent à l'espace des racines valides et pourraient être utilisées lors de la reconstruction. Cette erreur n'a pas d'impact sur l'erreur maximale sur la racine choisie calculée dans le tableau, néanmoins nos résultats montrent que cette erreur peut atteindre un maximum en simple précision de $3{,}4 \cdot 10^{-4} mm$. En partant de l'erreur maximale observée, nous nous proposons d'étudier l'impact potentiel sur la reconstruction dans la section suivante.

3.4.3.2 Calcul de l'erreur sur l'accès aux échantillons

Maintenant que nous avons évalué l'erreur numérique maximale, nous allons calculer son impact sur l'extraction des amplitudes. En effet, pour un signal à 2MHz (cas des trois premiers jeux de données), la période étant de 0,5 μ, le signal est sur-échantillonné pour être certain de ne pas perdre les pics. Dans ce cas, les échantillons seront séparés de 0,02 μs. La fréquence minimale d'échantillonnage (Fréquence de Nyquist) étant deux fois supérieure à la fréquence centrale, nous avons donc un facteur $\times 12{,}5$ de sur-échantillonnage, ce qui représente environ un ordre de grandeur.

Nous effectuons donc un calcul d'erreur sur le temps de vol en fonction de l'erreur évaluée. Pour cela, la méthode de recherche de zéros va devoir traiter un très grand nombre de cas d'angles et de distances. Nous avons donc posé notre problème tel que présenté sur la figure 3.13. De plus, il est à noter qu'afin de limiter les dimensions du problème, les paramètres ayant le moins d'impact ont été fixés.

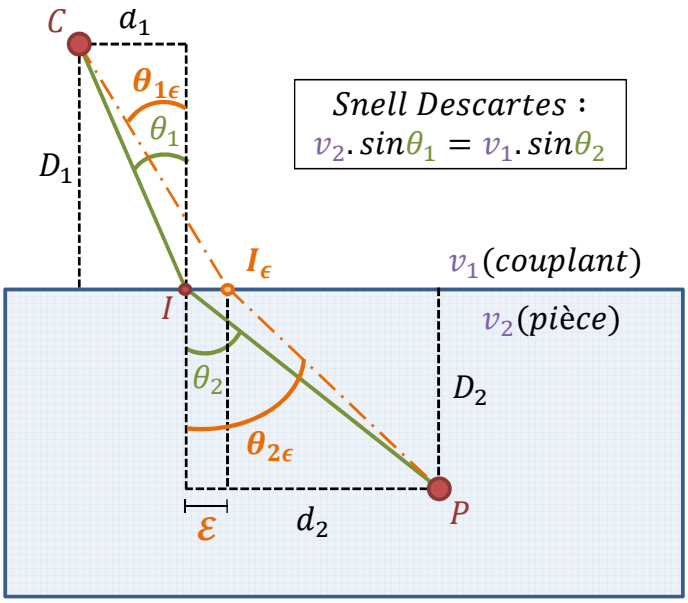


FIGURE 3.13 – Schématisation de l'impact de l'erreur sur les racines du polynôme dans le cadre de la reconstruction FTP.

- C : point correspondant à la position du traducteur
- P : point correspondant à la position du point de reconstruction
- I : point d'impact réel correspondant à l'égalité de Snell-Descartes entre C et P.
- ϵ : erreur numérique issue du calcul de résolution d'équations polynomiale)
- I_ϵ : point d'impact calculé via la méthode de résolution d'équations polynomiales (avec erreur numérique)
- θ_1, θ_2 : angles réels correspondant à l'égalité de Snell-Descartes entre C et P
- $\theta_{1\epsilon}, \theta_{2\epsilon}$: angles issus du point calculé I_ϵ
- $v1$, $v2$: vitesses des ultrasons, respectivement dans le milieu couplant et dans la pièce
- $D1$: distance de la projection orthogonale entre le point C et la surface de la pièce
- $D2$: distance de la projection orthogonale entre le point P et la surface de la pièce

Cherchant à calculer l'erreur sur une surface à l'intérieur de la pièce, la dépendance qui nous semble la plus intéressante est celle aux variables $d1$ et $D2$. Les autres données sont des constantes dont voici les valeurs :

- $D1$ = 25 mm
- $v1$ = 1483 $m.s^{-1}$
- $v2$ = 5900 $m.s^{-1}$
- ϵ = 3,4 $\cdot 10^{-4}$ mm

Les valeurs que nous cherchons sont les temps de vol. Connaissant les vitesses dans les deux milieux, il nous faudra donc déterminer les distances parcourues dans les cas des trajets exacts et avec erreur numérique. Nous pouvons les exprimer sous la forme suivante :

$$TempsVol(CP)_{Reel} = T_1 + T_2 \ \ avec \ \ T_1 = \frac{\overline{CI}}{v_1} \ , T_2 = \frac{\overline{IP}}{v_2} \tag{3.6}$$

$$TempsVol(CP)_{Calcul} = T_{1\epsilon} + T_{2\epsilon} \ \ avec \ \ T_{1\epsilon} = \frac{\overline{CI_\epsilon}}{v_1} \ , T_{2\epsilon} = \frac{\overline{I_\epsilon \, P}}{v_2} \tag{3.7}$$

A l'aide de l'égalité de Snell-Descartes et du théorème de Pythagore, nous pouvons exprimer les valeurs de T_1 et T_2 de la manière suivante :

$$T_1 = \frac{\sqrt{D_1^2 + d_1^2}}{v_1} \tag{3.8}$$

$$T_2 = \frac{D_2}{v_2 \cos\theta_2} \tag{3.9}$$

avec :

$$\theta_2 = \arcsin\left(\frac{d_1 v_2}{v_1\sqrt{D_1^2 + d_1^2}}\right) \tag{3.10}$$

Suivant le même raisonnement, nous obtenons directement $T_{1\epsilon}$, $T_{2\epsilon}$ et $\theta_{2\epsilon}$ en fonction des variables de notre problème :

$$T_{1\epsilon} = \frac{\sqrt{D_1^2 + (d_1 + \epsilon)^2}}{v_1} \tag{3.11}$$

$$T_{2\epsilon} = \frac{D_2}{v_2 \cos\theta_{2\epsilon}} \tag{3.12}$$

avec :

$$\theta_{2\epsilon} = \arcsin\left(\frac{(d_1 + \epsilon)v_2}{v_1\sqrt{D_1^2 + (d_1 + \epsilon)^2}}\right) \tag{3.13}$$

Nous faisons varier $d1$ entre 0 et 10 mm et $D2$ entre $0{,}1$ et 500 mm puisque ces valeurs couvrent un large ensemble de cas réels (particulièrement en profondeur où l'on traite rarement des pièces de profondeur supérieures à 500 mm).

La figure 3.14 présente la valeur de l'erreur en fonction de $d1$ et $D2$. On constate que l'erreur est linéairement proportionnelle à $D2$ et qu'elle croît de manière quadratique par rapport à d1. Il est à noter que lorsque $d1 > 6$ mm, on dépasse l'angle critique pour la relation de Snell-Descartes (l'erreur est affichée comme nulle, mais ces cas n'existent pas en pratique). L'erreur croît linéairement avec la profondeur et atteint, pour une profondeur de 500 mm, une valeur de

$6{,}5 \cdot 10^{-5}\ ms$. Cette valeur d'erreur est à doubler puisqu'ici, le calcul est fait pour un temps de vol en émission auquel s'ajoute un temps de vol en réception. Cette valeur est à comparer avec la temps entre deux échantillons de signal qui est de $2{,}0 \cdot 10^{-5}\ ms$. L'erreur peut donc dépasser l'échantillonnage des signaux, ce qui pourrait s'avérer problématique lors de la reconstruction, entraînant des artefacts.

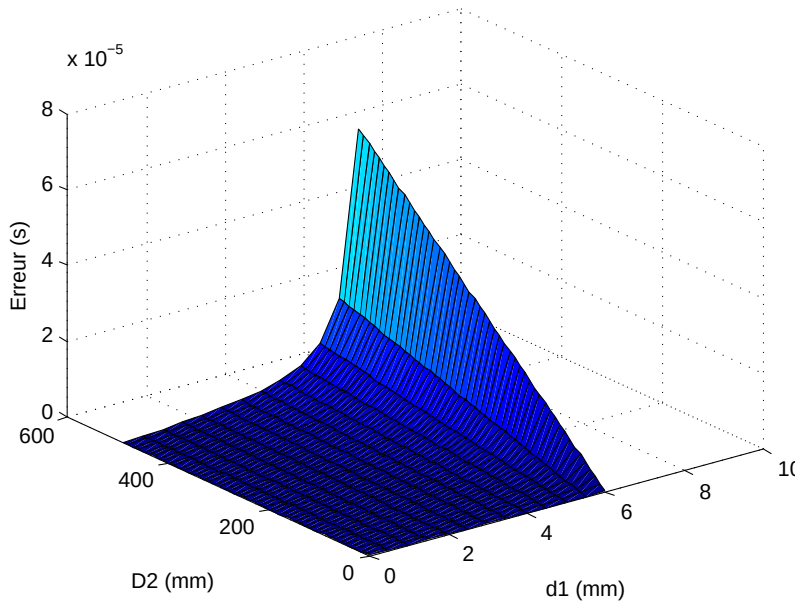


FIGURE 3.14 – Evaluation de l'erreur sur les temps de vol lors de la reconstruction FTP.

Nous avons vu que la résolution d'équations polynomiales en F32 pouvait être source d'erreurs, même si l'on ne prend pas en compte les cas où la méthode retourne des racines réelles plutôt que complexes. De plus, en considérant une erreur relativement faible sur la précision des racines, on a vu que celle-ci pouvait avoir un impact important sur l'évaluation du temps de vol, ce qui pourrait impliquer dans certains cas des artefacts lors de la reconstruction. Nous pouvons donc conclure que la précision obtenue en F32 n'est pas utilisable de manière générique pour l'algorithme FTP. Il va être nécessaire de se limiter en termes de dimensions de pièces. Nous allons maintenant affiner ces résultats en nous intéressant à la précision obtenue de manière générale, en utilisant un logiciel qui permet d'évaluer les erreurs d'arrondis : CADNA.

3.4.3.3 Exploitation du logiciel CADNA pour évaluer les erreurs d'arrondis

.

Lors des développements, nous avons rencontré plusieurs problèmes de précision, se traduisant sous forme d'artefacts sur les images reconstruites par l'algorithme FTP. Afin de déterminer leur origine, nous avons entrepris d'utiliser une méthode plus formelle que celle présentée dans la section précédente.

Les approches d'évaluation d'erreurs d'arrondi

L'arithmétique des nombres flottants étant une approximation de l'arithmétique exacte, une

erreur d'arrondi est introduite par chaque opération ou affectation. Il existe plusieurs types d'approches permettant d'évaluer les erreurs d'arrondi [Chesneaux, 1995] :

- l'analyse régressive ou inverse,
- l'analyse directe,
- l'approche déterministe,
- l'approche probabiliste.

Les deux premières approches sont basées sur une analyse statique tandis que les deux dernières sont effectuées directement lors de l'exécution d'un code, exploitant la précision finie de la machine utilisée. L'analyse régressive est limitée à l'obtention de la majoration de l'erreur globale, tandis que l'analyse directe semble intéressante, mais est peu répandue. L'approche déterministe est, elle aussi, très intéressante, car elle permet d'encadrer dans un intervalle le résultat d'un calcul. Un intervalle correspondant à l'erreur est attribué à chaque opération et c'est la somme de ces intervalles qui permet d'encadrer le résultat. L'arithmétique d'intervalle est très utilisée, mais nécessite de manière générale de reprendre l'algorithme utilisé pour éviter une expansion de l'intervalle vers l'infini. Cette approche est donc coûteuse.

L'approche probabiliste permet de déterminer l'erreur d'arrondi engendrée lors de l'exécution d'un programme sur une machine quelconque. Chaque erreur d'arrondi est considérée comme une variable aléatoire ce qui permet de prendre en compte les compensations d'erreurs d'arrondis. Si la méthode est plus simple que les approches citées précédemment, elle n'est pas sans inconvénient : l'erreur obtenue est dépendante d'une probabilité.

Basée sur la méthode CESTAC qui permet d'estimer la propagation d'erreurs d'arrondi de tout résultat informatique, le LIP6 de l'Université Pierre et Marie Curie développe un logiciel permettant d'exploiter de manière relativement simple et transparente les erreurs d'arrondi d'un code Fortran, C ou C++. Les opérateurs mathématiques et d'affectation sont surchargés pour prendre en compte l'arithmétique stochastique et de donner une évaluation de la précision des valeurs flottantes. Les valeurs flottantes sont évaluées à l'aide de trois valeurs, dit triplet stochastique. A chaque opération, chaque membre du triplet est évalué fonction d'un mode d'arrondi déterminé aléatoirement. Ce changement de mode d'arrondi permet à certaines opérations d'être plus justes et à d'autres d'être plus fausses, de telle sorte que les valeurs du triplet soit une probabilité différente de la valeur réelle.

L'intégration dans un code de calcul est relativement aisée et ne demande qu'à remplacer le type `float` ou `double` par un type CADNA `float_st`. Deux primitives sont ensuite mises à disposition pour indiquer le début de la zone à tester et la fin. Une fois la section terminée, CADNA affiche la somme des différentes erreurs d'arrondi rencontrées. C'est à l'utilisateur de remonter à la source des erreurs à l'aide d'un debugger pour comprendre d'où viennent les différents problèmes.

Utilisation de CADNA dans le cadre de l'algorithme de Laguerre

Nous avons intégré CADNA dans notre implémentation de Laguerre afin de mesurer la précision des résultats suite aux multiples erreurs d'arrondis entraînées par les nombreuses itérations de l'algorithme. Pour les reconstructions de cas plans et cylindriques, nous nous sommes contentés de faire une validation visuelle en comparant nos images de sortie. Allant jusqu'au degré 10, nous n'avons pas détecté de problème majeur parmi ces polynômes puisque aucun artefact n'est à noter.

En revanche, c'est sur le jeu de polynômes utilisé en section 3.4.3.1 et issu d'une reconstruction FTP d'une pièce torique que nous avons rencontré des problèmes plus intéressants. En degré, 16, les coefficients des polynômes atteignent une dynamique qui devient problématique pour certaines résolutions. Nous allons illustrer le cas avec le polynôme suivant :

$$\begin{aligned} P(x) = {} & (0.732921501129208 \cdot 10^{-03})x^{16} + (-0.672281823587994)x^{15} + \\ & (-0.804337686237056 \cdot 10^{03})x^{14} + (0.675419108059512 \cdot 10^{06})x^{13} + \\ & (0.183294652820367 \cdot 10^{09)}x^{12} + (-0.133868348361080 \cdot 10^{12})x^{11} + \\ & (0.156823736024058 \cdot 10^{14)}x^{10} + (-0.137120196956186 \cdot 10^{14})x^{9} + \\ & (0.364797343912177 \cdot 10^{15)}x^{8} + (-0.791410972627428 \cdot 10^{14)}x^{7} + \\ & (0.143226901768207 \cdot 10^{16)}x^{6} + (-0.389889678919520 \cdot 10^{14)}x^{5} + \\ & (0.576605068893468 \cdot 10^{13)}x^{4} + (-0.688809764675349 \cdot 10^{10)}x^{3} + \\ & (-0.142834557867462 \cdot 10^{08)}x^{2} + (0.989266544627827 \cdot 10^{04)}x^{1} + \\ & 0.116699379731214 \cdot 10^{02} \end{aligned} \tag{3.14}$$

Lorsque l'on va effectuer des opérations d'addition ou de soustraction entre des nombres de dynamique si différente, on va s'exposer à des erreurs d'*absorption*. En effet, lorsque l'on ajoute un nombre flottant A à un nombre flottant B, et que A est très supérieur à B, il est possible que la valeur de B soit ignorée par le calcul, entraînant une erreur qui peut s'avérer importante dans la suite des calculs.

De la même manière, si on soustrait un nombre A à un nombre B, avec A très proche de B, il est possible d'avoir une erreur dite de *cancellation*. Le résultat est donc très petit par rapport aux deux opérandes, et il est possible que la valeur réelle soit complètement ignorée, entraînant une *cancellation catastrophique*.

Sur le polynôme ci-dessus, on peut observer que les coefficients vont, pour le plus petit de 0,73E-3 au plus grand à 0,14E16 pour respectivement l'ordre 16 et l'ordre 6. Cela représente un écart de 19 chiffres significatifs. Les opérations réalisées entre ces nombres risquent d'entraîner des erreurs d'absorption.

L'exécution avec CADNA nous montre immédiatement une erreur très importante sur la première racine trouvée nous affichant le résultat suivant : `@.0`. Ce symbole est utilisé pour dire que le résultat obtenu n'a aucun chiffre significatif et on parle de zéro computationnel dans ce cas. Cela veut tout simplement dire que l'ensemble des racines trouvées par la suite n'ont elles-même aucune validité du fait de l'utilisation de cette racine pour factoriser le polynôme. Nous allons donc nous concentrer sur ce résultat pour essayer de comprendre d'où vient le problème.

Nous avons tracé la perte de précision en observant le comportement de la racine à chaque itération de Laguerre. La valeur initiale de la racine est 0 dans la mesure où nous n'avons pas d'apriori particulier des racines à trouver. Voici les valeurs obtenues pour les différentes itérations :

```
Itération 0 : x = 0.0000000E+000
Itération 1 : x = 0.5780986E+003
```

```
Itération 2 : x = 0.8852555E-003
Itération 3 : x = 0.968208E-003
Itération 4 : x = 0.974948E-003
Itération 5 : x = 0.97507E-003
Itération 6 : x = 0.973E-003
Itération 7 : x = 0.10E-002
Itération 8 : x = @.0
```

Nous pouvons observer que le critère de Adams n'est pas suffisant dans ce cas et ne permet pas de sortir suffisamment tôt. La valeur sur laquelle Laguerre converge est la suivante : (0.0009749561892 , 0) (obtenue en F64). A l'itération 4, la valeur est donc précise à 4 chiffres significatifs près ce qui est suffisant et n'a perdu à ce moment qu'un seul chiffre significatif sur sa valeur initiale ce qui est très correct. En revanche, le fait de continuer à itérer entraîne la perte d'un chiffre significatif à l'itération 5, un à l'itération 6, un à l'itération 7 puis les deux derniers à la dernière itération.

Nous allons donc essayer de déterminer d'où viennent ces erreurs et s'il est possible de les minimiser en utilisant le debugger comme expliqué précédemment. Le rapport d'erreur du run précédent nous précisant un grand nombre d'erreurs, nous sélectionnons uniquement les *cancellations* à l'aide de la primitive `cadna_init`. Les deux *cancellations* détectées se trouvent dans l'évaluation de Horner. En effet, Laguerre cherchant une racine qui annule le polynôme, l'évaluation de celui-ci arrive logiquement à une valeur proche de zéro quand la racine est suffisamment précise. C'est exactement ce qui se produit ici, entraînant une erreur de *cancellation* pour l'opération suivante et les valeurs des triplets stochastiques suivants :

```
b += x*b + poly[j]
```

avec

```
b = {x = -0.751752734, y = -0.751753151, z = -0.751752555,
             accuracy = 6}
x = {x = 0.000974948634, y = 0.000974947994, z = 0.000974948925,
             accuracy = 6}
poly[j] = {x = 0.00073292153, y = 0.00073292153, z = 0.00073292153,
             accuracy = 7}
```

et le résultat de l'opération

```
b = {x = 1.22236088e-09, y = 1.3387762e-09, z = 1.16415322e-09,
             accuracy = -1}
```

Cette erreur ne devrait pas être importante si l'on réussit à la détecter et sortir de la boucle de Laguerre. Cependant, les critères que nous utilisons ne sont pas capables de le faire et nous continuons le calcul de Laguerre avec cette valeur de b très petite et surtout fausse. Heureusement pour nous, le delta calculé par Laguerre pour itérer est d'autant plus faible que cette valeur l'est, ce qui permet, bien que b soit faux, de ne pas modifier la valeur de la racine de manière significative. On sort donc de la boucle à l'aide du critère de Adams, mais seulement 3 itérations plus tard. Ces erreurs de *cancellation* ne sont donc pas nécessairement problématiques pour nous.

L'ensemble des autres erreurs, multiplication, branchement et division, arrivent après cette 5e itération. Cela nous montre que finalement, l'ensemble des erreurs détectées par CADNA ne

sont a priori pas critiques et ne devraient pas avoir d'impact sur notre racine. Si maintenant, on exécute le code sans CADNA, on observe que la valeur de racine obtenue en F32 est identique jusqu'au dernier chiffre significatif à celle obtenue en F64 :

```
Racine F32 :  0.000974956
Racine F64 :  0.0009749561892
```

Etant donné que l'étude a été réalisée sur la première racine trouvée, il n'est pas forcément surprenant d'obtenir un résultat identique, d'autant plus que les calculs ont été réalisés en réel. Dans le cas de racines complexes, la quantité de calcul est significativement supérieure et donc source de plus d'erreurs d'arrondis. Les factorisations nécessaires à la recherche des zéros amplifient donc ces erreurs. En pratique, sur les polynômes parmi les plus mal conditionnés de notre jeu de données, nous perdons jusqu'à 3 chiffres significatifs (sur un total de 7).

CADNA nous permet de formaliser en partie le fait que la simple précision suffit de manière générale, et ce même pour des polynômes de degré 16. La raison pour laquelle la validation n'est que partielle est que CADNA ne permet de valider qu'un jeu de donnée numérique donné. Il existera toujours des polynômes dont le conditionnement les rend très complexes à résoudre, tel que le polynôme de Wilkinson [Wilkinson, 1959] par exemple. Nous pouvons tout de même conclure que nous pouvons utiliser la simple précision pour l'algorithme de résolution de Laguerre.

3.4.3.4 Conclusions

Dans cette partie, nous avons essayé de valider l'utilisation de la simple précision pour réaliser le calcul de FTP basé sur la méthode de Laguerre. Nous avons utilisé deux approches de validation.

La première nous a permis, en déterminant le paramètre jouant directement sur la précision, de montrer qu'à partir d'une certaine dimension de pièce, l'erreur d'arrondi pouvait être source d'artefact. Même si ce type de pièce est relativement rare, la généralisation de l'utilisation de la méthode en simple précision peut s'avérer risquée.

La seconde approche est basée sur l'utilisation du logiciel CADNA et son modèle stochastique. Nous avons directement évalué la précision des racines obtenues pour des polynômes de degré 16 utilisés dans les cas toriques. Nous avons pu voir que la précision obtenue est suffisante pour les calculs de FTP puisque la perte d'un maximum de trois chiffres significatifs a été observée. Nous avons pu aussi observer que le critère de Adams permet, dans la grande majorité des cas, de sortir suffisamment tôt par rapport à la précision machine. Les rares cas où quelques itérations supplémentaires sont réalisées, les erreurs commises sont si faibles qu'elles n'entachent pas la valeur de la racine.

Ces résultats ayant été obtenus postérieurement à la majeure partie des développements et des benchmarks, les sections suivantes traiterons principalement de résultats double précision. Cependant, un développement SIMD a été réalisé, et au travers de cette implémentation nous étudierons les performances de FTP en cas d'utilisation de simple précision.

3.5 Etude de la parallélisation sur GPU

Dans cette section, nous allons détailler les implémentations GPU basées sur les approches discutées dans la section 3.3.7, puis dans un second temps, nous effectuerons un ensemble de benchmarks pour évaluer les performances des GPU sur CUDA et OpenCL sur les différentes fonctionnalités que nous avons implémentées pour la méthode de reconstruction FTP. L'implémentation OpenCL étant un portage direct de l'implémentation CUDA, nous présentons les deux directement dans la sous-section qui suit.

3.5.1 Implémentations CUDA et OpenCL

Nous allons présenter les implémentations GPU. Le modèle de programmation de ces architectures est SIMT, ce qui implique l'exécution d'une même instruction par un groupe de threads de manière synchrone. Nous allons donc essayer d'adapter au mieux l'algorithme à ce modèle, en prenant en considération les dimensions de nos paramètres pour utiliser au mieux la mémoire partagée du GPU.

3.5.1.1 Base des implémentations GPU

Comme nous l'avons vu dans la section 3.3.7, nous allons paralléliser sur les pixels de l'image afin de bénéficier de la réutilisation des temps de vol. Sur GPU, deux niveaux de parallélisation sont accessibles. Le premier se trouve au niveau des threads et le second sur les blocs de threads. Une première approche serait donc de calculer un pixel par thread.

Comme nous l'avons vu pour l'algorithme 4, il va être nécessaire de stocker l'ensemble des temps de vol en émission vers un point de la zone de calcul pour effectuer ensuite l'extraction et la sommation des amplitudes. De plus, comme nous l'avons vu précédemment, nous nous intéressons à des traducteurs dont le nombre d'éléments est compris entre 16 et 256. Nous allons donc avoir besoin de stocker jusqu'à $256\times$ `sizeof(double)` $= 2048$ octets. Dans le cas de l'association "1 thread / 1 pixel", il serait donc nécessaire, pour un bloc de 32 threads, une dimension très restreinte, d'avoir 64 Ko de mémoire partagée. Ce n'est pas le cas, étant donné que l'architecture Fermi permet d'obtenir jusqu'à 48 Ko. Ce niveau de parallélisation n'est donc pas approprié.

Nous allons donc nous intéresser à l'association d'un bloc de threads par pixel. Par rapport à la première association. Le nombre de threads n'a ici aucun impact sur la quantité de mémoire partagée ; nous allons voir par la suite leur manière de se partager les calculs. Pour un traducteur de 256 éléments, il va être nécessaire de stocker à nouveau 2 Ko de temps de vol ($256\times$ `sizeof(double)` $= 2Ko$). Cette fois-ci, notre bloc de threads va pouvoir profiter de la mémoire partagée. De plus, comme nous l'avons vu dans le chapitre 1, le GPU est capable d'ordonnancer des *warps* de threads provenant de plusieurs blocs, à condition d'avoir suffisamment de ressources. Dans le cas présent, il serait possible de saturer le nombre maximum de blocs de threads du multiprocesseur (8, donc $8 \times 2Ko = 16Ko$), sans pour autant saturer la mémoire partagée. Ce type de parallélisme semble donc permettre de charger de manière efficace le GPU.

Pour cette parallélisation, nous allons devoir partager les calculs entre les threads pour obtenir de bonnes performances. Ce n'est pas un problème dans la mesure où les calculs de temps

de vol sont du même ordre de grandeur que le dimensionnement du bloc de threads et que l'extraction/sommation est, elle, de dimension bien plus élevée (puisqu'en $O(N_t^2)$). Le calcul des temps de vol, effectué dans la première boucle de l'algorithme 4, est partagée entre les threads tel que présenté dans le tableau 3.4. Le cas présenté ici est pour un bloc de threads de 6 threads et un traducteur de 10 éléments. Les 6 premiers threads exécutent les calculs de temps de vol des 6 premiers éléments en parallèle, puis dans un second temps les éléments 6 à 9 sont traités par les threads 0 à 3. Les threads 4 et 5 restent passifs. Dans le cas où le nombre d'éléments du traducteur n'est pas une puissance de deux, la charge entre les threads ne sera pas idéale entraînant une légère perte d'efficacité.

Tableau 3.4 – Ordonnancement des threads du GPU pour le calcul des temps de vol. A chaque élément de capteur, un temps de vol est calculé. Pour l'exemple ici, le travail est réparti entre 6 threads pour un traducteur de 10 éléments.

Thread ID	0	1	2	3	4	5
Elément 0	x					
Elément 1		x				
Elément 2			x			
Elément 3				x		
Elément 4					x	
Elément 5						x
Elément 6	x					
Elément 7		x				
Elément 8			x			
Elément 9				x		

Dans le cas d'une association d'un bloc de threads par pixel, et d'un capteur de 128 éléments, on aura tendance à vouloir utiliser 128 threads par bloc pour avoir une association 1 pour 1. Ceci reste contraignant pour des configurations plus petites, lorsqu'on aura un traducteur 32 éléments par exemple, où 32 sera une taille de bloc relativement sous dimensionnée.

Une fois les temps de vol calculés, il est nécessaire d'effectuer une synchronisation des threads du bloc afin de vérifier que tous ont terminé leur calcul pour pouvoir passer à la boucle d'extraction et de sommation des amplitudes.

Pour cette seconde boucle, les extractions/sommations sont partagées comme pour la première boucle. La différence principale réside dans le fait que l'ensemble des calculs effectués est sommé dans la valeur du pixel. Etant donné que tous les threads du bloc accèdent à ce pixel, il est donc nécessaire de synchroniser l'accès à cet emplacement mémoire en utilisant un *atomicAdd*. Etant donné que les threads extraient plusieurs amplitudes (nombre de threads par bloc / N_t^2), ces sommations sont effectuées dans un registre, et la réduction à l'aide de l'instruction atomique uniquement effectuée une fois le travail des threads terminé.

Les détails d'implémentations présentés dans cette section sont les grandes lignes des deux implémentations réalisées. Nous allons maintenant présenter leurs spécificités.

3.5.1.2 *Exhaust* : implémentation sans interpolation des temps de vol

Les détails d'implémentation présentés ci-dessus nous permettent d'apporter une optimisation supplémentaire pour obtenir l'implémentation que nous appellerons *Exhaust*. En effet, la mémoire partagée du GPU est utilisée, mais il reste une certaine partie de cette mémoire disponible, même pour les traducteurs de grandes dimensions. Nous proposons donc d'ajouter un niveau de parallélisation au sein du bloc de threads afin d'apporter les optimisations suivantes :

1. possibilité d'utiliser plus de mémoire partagée,
2. réduction de la concurrence des accès atomiques,
3. localisation des accès aux signaux permettant d'obtenir des gains grâce au cache.

Pour le point (1), si plusieurs pixels par bloc sont traités, il va falloir stocker les temps de vol de ces pixels. Le point (2) est aussi relativement immédiat puisque si on traite plusieurs pixels, la réduction finale portera sur ces différents pixels et donc la concurrence sera d'autant plus faible qu'on traite plusieurs pixels. Enfin, concernant le point (3), les threads qui accèdent à des signaux pour différents pixels vont être synchronisés, en utilisant la notion du warp, afin d'accéder au même signal en même temps, et en se basant sur que le fait qu'entre deux pixels différents, le temps de vol calculé sera proche, et donc que les amplitudes extraites pour les différents pixels seront proches en mémoire. Cette optimisation permettra d'obtenir une meilleure efficacité du cache L1 des GPU Fermi.

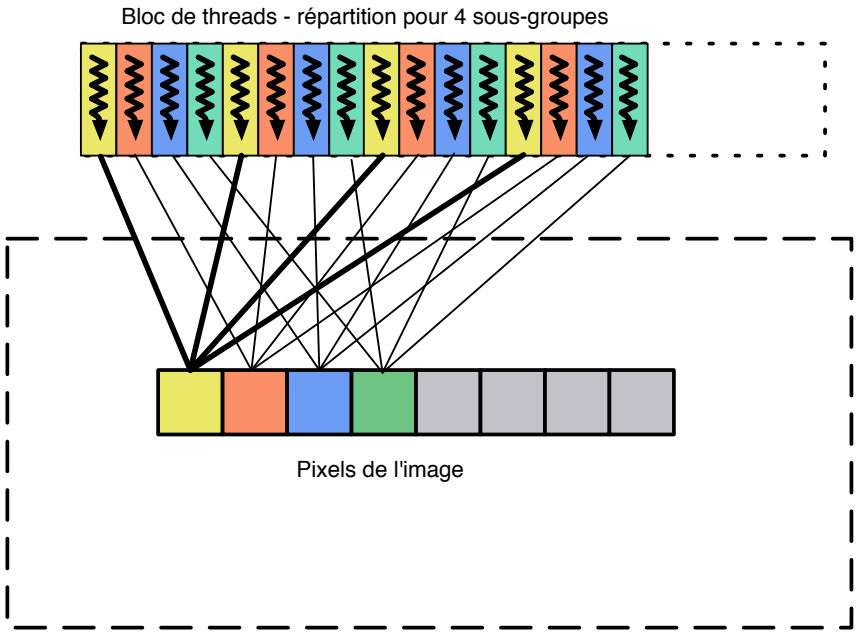


FIGURE 3.15 – Schéma présentant l'ordonnancement des threads d'un bloc pour l'implémentation *Exhaust* avec plusieurs pixels par bloc de threads lors de la boucle d'extraction/sommation des amplitudes. Cet ordonnancement permet entre autres à plusieurs threads d'accéder au même signal en même temps permettant d'obtenir des accès cache au lieu d'accès en globale *Exhaust*if pour une implémentation avec un bloc de threads par pixel de l'image.

Pour illustrer nos propos, la figure 3.15 présente l'ordonnancement des accès pour un bloc de threads traitant différents pixels. L'ordre est important puisqu'il permet à des threads voisins de traiter différents pixels en parallèle et donc d'accéder aux même signaux en même temps comme nous venons de le dire. Ce type d'optimisation dépend bien évidemment de la résolution de l'image, qui sera d'autant plus efficace que celle-ci sera fine par rapport à l'échantillonnage des signaux. Par exemple, pour le jeu de données DATA 1, l'espacement entre deux pixels est de $0{,}2\ mm$ et l'ordre de grandeur de la différence de temps de vol entre ces points est de $0{,}06\ \mu s$. Etant donné que la fréquence d'échantillonnage est de $0{,}02\ \mu s$, l'accès à deux échantillons se fera avec un pas de 3 échantillons.

Tableau 3.5 – Quantité de mémoire partagée utilisée pour 8 blocs de threads par multiprocesseur (cas idéal pour l'ordonnancement sur le GPU) en fonction de la taille du traducteur ultrasons et du nombre de pixels traités par bloc de threads. Les valeurs grisées ne sont pas utilisables, et celles en italique sont des optimums.

		# pixels / bloc de threads			
nombre d'éléments	*shared*/pixel (Ko)	1	2	4	8
32	256	2048	4096	8192	16384
64	512	4096	8192	16384	*32768*
128	1024	8192	16384	*32768*	65536
256	2048	16384	*32768*	65536	131072

Nous constatons que pour cette implémentation, selon la taille de bloc, et le nombre de pixels par blocs traités, la quantité de mémoire partagée utilisée va être variable. Le tableau 3.5 présente différentes combinaisons de tailles de traducteurs et de nombre de pixels traités par un bloc de threads. On considère que l'on cherche à pouvoir exécuter le maximum de blocs de threads par multiprocesseur en espérant obtenir le meilleur ordonnancement possible. Etant donné que le nombre maximum de blocs exécutable en parallèle sur un multiprocesseur des GPU Nvidia est 8, on multiplie la taille de mémoire *shared* utilisée par 8. On peut constater que toutes ne sont pas possibles telles que celles grisées dans le tableau étant donné que le GPU ne dispose que de 48Ko de mémoire *shared* au maximum par multiprocesseur. Ce tableau nous permet aussi de déterminer des optimums locaux (valeurs en italiques). Il est à noter qu'obtenir 48Ko serait évidemment mieux, mais nous allons rester dans nos cas de tests sur des puissances de 2. 32 Ko est donc la valeur retenue. Nous validerons cette approche dans les benchmarks.

3.5.1.3 *Interp* : Implémentation avec interpolation des temps de vol

L'implémentation avec interpolation des temps de vol est basée sur l'algorithme 5. La différence principale par rapport à l'implémentation *Exhaust* est que la parallélisation ne va plus se faire sur les pixels de l'image mais sur les tuiles du maillage d'interpolation. Il va donc être nécessaire de stocker 4 tableaux de temps de vol au lieu d'un. Au niveau du calcul des temps de vol, la répartition des calculs entre les threads est identique à celle utilisée pour l'implémentation *Exhaust*.

Concernant la boucle d'extraction et de sommation des amplitudes, le principe de localité est lui aussi repris sur l'implémentation *Exhaust*. Les threads calculant les amplitudes des pixels de la tuile vont être répartis de sorte qu'ils travaillent sur les mêmes couples émetteur-récepteur en parallèle. Dans ce cas, le nombre de sous-groupes ne va pas avoir d'impact sur la quantité de mémoire partagée utilisée. En effet, les temps de vol sont calculés aux coins de la tuile et sont donc indépendants des groupes de pixels. Il va donc être possible d'utiliser un parallélisme plus important que pour l'implémentation *Exhaust*. Cette optimisation est actuellement 1D, mais il serait possible de l'étendre en 2D.

A l'heure actuelle, nous n'utilisons pas de recouvrement entre plusieurs tuiles. Il serait intéressant de l'utiliser et particulièrement pour des traducteurs de petite taille où la mémoire partagée est moins utilisée. Cela permettrait d'augmenter l'efficacité de l'algorithme dans ces cas. Ce recouvrement permettrait aussi d'éviter de recalculer des points. En effet, pour deux tuiles côte à côte, seuls 6 points seraient nécessaires. Néanmoins, cette optimisation est très limitée dans la mesure où l'interpolation en elle-même réduit déjà considérablement la quantité de calculs.

Passons maintenant à l'analyse des différents benchmarks effectués sur les deux implémentations présentées ci-dessus.

3.5.2 Analyse des performances des GPU

3.5.2.1 Impact de l'association bloc de threads/nombre de pixels par bloc

Nous cherchons à obtenir les paramètres de dimensionnement de *kernel* optimaux pour nos deux implémentations *Exhaust* et *Interp*. Les paramètres sont pour une taille de traducteur donné, la taille du bloc de threads et le nombre de pixels traités par bloc. On va donc faire varier ces paramètres sur des intervalles correspondants à des valeurs classiques sur GPU telles que présentée par [Kirk and Hwu, 2010]. Concernant les tailles de bloc, l'intervalle classique sur GPU est $[32; 256]$. Ces valeurs permettent d'obtenir un chargement suffisant du GPU si plusieurs blocs peuvent être exécutés en parallèle sur les multiprocesseurs du GPU. Quant aux valeurs du nombre de pixels par bloc, nous les faisons varier entre 1 et 4 pour l'implémentation *Exhaust*, et 1 et 16 pour l'implémentation *Interp*. Dans le cas de l'implémentation *Exhaust*, ce dernier paramètre joue directement sur la quantité de mémoire partagée utilisée, alors que pour l'implémentation *Interp*, cela n'a pas d'impact et cela permet donc d'utiliser une valeur plus grande. La valeur de 16 est déterminée en fonction du nombre de pixels par tuile, qui se trouve être de 14×14 pour la résolution utilisée sur DATA 1.

Nous avons effectué un paramétrage à l'aide du jeu de données DATA 1, sur C2070. Les figures 3.16 et 3.17 présentent les performances des implémentations CUDA *Exhaust* et *Interp* sur les paramétrages proposés ci-dessus.

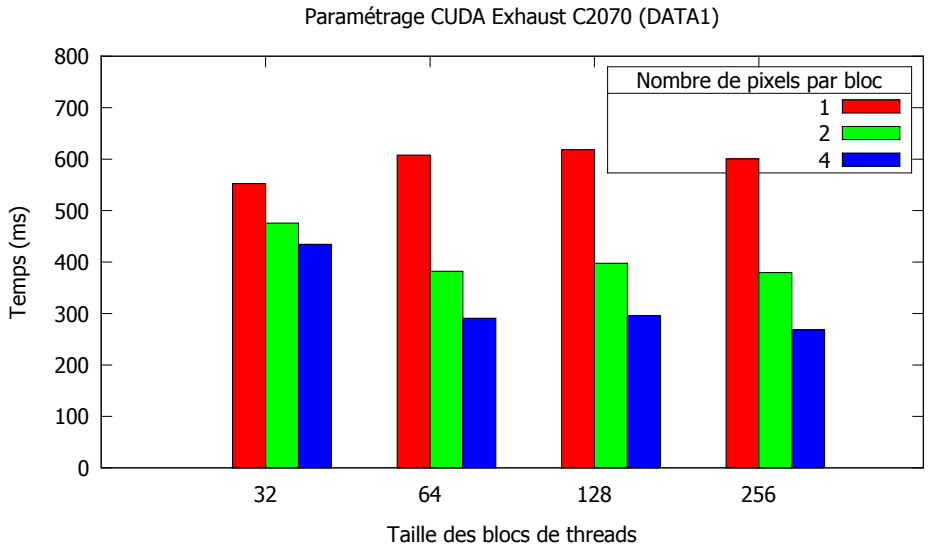


FIGURE 3.16 – Paramétrage du noyau de calcul *Exhaust* pour l'implémentation CUDA sur C2070.

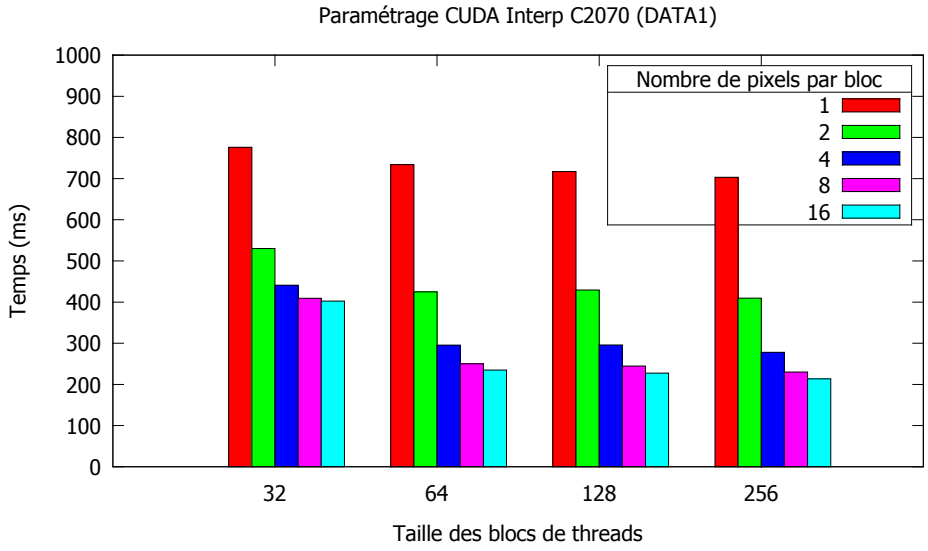


FIGURE 3.17 – Paramétrage du noyau de calcul *Interp* pour l'implémentation CUDA sur C2070.

On constate sur la figure 3.16 que plusieurs valeurs sont optimales, mais à chaque fois pour le plus grand nombre possible de pixels par blocs. On peut constater des écarts de performances allant jusqu'à un facteur $\times 2$. L'association 256/4 permet d'obtenir les meilleures performances pour l'implémentation *Exhaust*. Cependant, les cas 64/4 et 128/4 sont très proches, avec des variations à la baisse de moins de 2%. Nous allons donc choisir d'effectuer nos benchmarks avec le couple 64/4 et ce afin de garder une bonne efficacité pour les cas de jeux de données de petites dimensions, pour lesquels cette taille de bloc est plus adaptée.

Pour l'implémentation *Interp*, on peut constater le même principe sur la figure 3.17. C'est l'association 256/16 qui permet d'obtenir les meilleures performances. En revanche ici, le fait d'augmenter le nombre de pixels par blocs n'implique pas une augmentation de l'utilisation de mémoire *shared* (c.f. section 3.5.1.3), on peut donc aller jusqu'à 16 pixels traités par un même bloc. Grâce à cela on peut constater un écart de performances allant jusqu'à un facteur $\times 3{,}5$. Pour les mêmes raisons que précédemment, nous allons préférer le couple 128/16 à 256/16 pour des raisons d'efficacité, mais la différence de performances n'est plus significative à ce niveau.

Sachant que le dimensionnement des blocs de threads en CUDA ou OpenCL est déjà une thécmdmatique d'optimisation [Torres et al., 2012], l'ajout de l'optimisation architecturale permettant un sous-découpage de bloc amène des différences de performances d'autant plus importantes entre les différentes configurations. Nous avons synthétisé les différentes associations optimales que nous avons obtenues dans le tableau 3.6 pour les différentes implémentations et GPU utilisés. Ces valeurs ont été obtenues en moyennant les performances sur les quatre jeux de données avec les configurations par défaut. Nous pouvons nous permettre d'effectuer cette moyenne dans la mesure où le nombre d'éléments du capteur est constant entre les jeux de données.

On remarque qu'entre CUDA et OpenCL, les paramètres ne sont pas identiques à architecture égale. Malgré cela, les valeurs sont très proches. Si par exemple, les paramètres obtenus en CUDA sur la C2070 sont utilisés sur l'implémentation OpenCL, l'écart de performances n'est que de 1,3%. Les paramètres de dimensionnement des *kernels* présentés dans le tableau précédent seront utilisés dans la suite des benchmarks.

Tableau 3.6 – Association optimale entre le nombre de threads par bloc et le nombre de pixels traités par bloc pour chaque Implémentation.

	Implémentation *Exhaust*		Implémentation *Interp*	
	taille de bloc	nombre de pixels	taille de bloc	nombre de pixels
C2070 CUDA	64	4	128	16
GTX 580 CUDA	64	4	128	16
C2070 OpenCL	64	4	256	8
GTX 580 OpenCL	64	4	128	16
HD 6970 OpenCL	64	4	128	8

3.5.2.2 Impact de la résolution de l'image

Bien que nos configurations par défaut aient une résolution fixée, basée sur une valeur moyenne du besoin de l'utilisateur de l'algorithme FTP, nous avons effectué une étude de performances pour différentes tailles d'image et observé le comportement des différentes implémentations. Nous présentons les résultats sur les jeux de données DATA 1 et DATA 4. Les quatre figures, 3.18 à 3.21 présentent les résultats pour les deux jeux de données et les deux méthodes de calcul. Les temps affichés sont normalisés en fonction du nombre de pixels de l'image. Cela permet d'obtenir un temps de calcul par pixel. Deux points importants sont à noter :

- seule la résolution pixel varie, permettant au calcul par interpolation d'être logiquement de plus en plus performant lorsque la résolution grandit,
- les courbes obtenues pour les jeux de données DATA 1 et DATA 4 ne sont pas directement comparable à résolution pixel égales car le maillage sur lequel le calcul est effectué en interpolation dépend du ratio entre la maille physique et la maille pixel. Pour le jeu de données DATA 4, le ratio est inférieur au jeu de données DATA 1 à résolution pixel égale. Cela permet d'obtenir des temps de reconstruction plus adaptés au benchmark et à l'utilisateur qui évitera d'aller chercher des résolutions trop importantes pour des reconstructions qu'il prévoit comme déjà très longues.

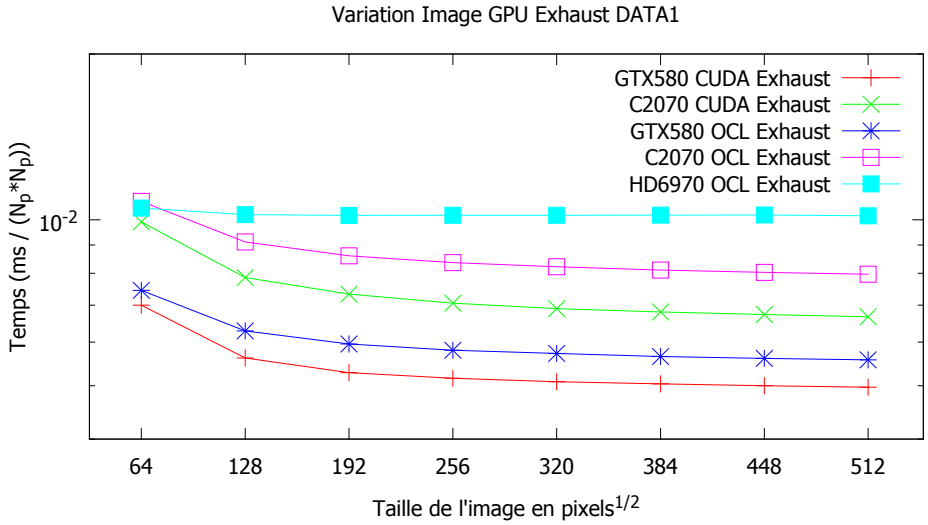


FIGURE 3.18 – Temps de calcul normalisé pour l'implémentation *Exhaust* sur GPU pour le jeu de données DATA 1 et pour des tailles d'images de 64×64 à 512×512.

Le jeu de données DATA 1 nécessite le moins de calculs. En effet, dans le cas d'une surface plane, la résolution d'équations polynomiales se fait comme on l'a vu sur un polynôme de degré 4. On constate sur la figure 3.18 que le temps de calcul normalisé devient rapidement constant à partir d'une dimension d'image proche de 200×200, et ce, quelle que soit l'implémentation ou l'architecture. On notera que la courbe de la HD6970 est la plus constante des cinq courbes, ne bénéficiant pas d'amélioration de l'efficacité pour de plus grands traducteurs. Ses performances sont légèrement en retrait des cartes Nvidia. De manière générale, on peut simplement constater qu'à partir d'une certaine quantité de pixels, l'implémentation devient linéairement dépendante du nombre de pixels de l'image. Sur la figure 3.19, on peut constater le même comportement, avec ici une HD6970 au même niveau de performances que la Tesla C2070.

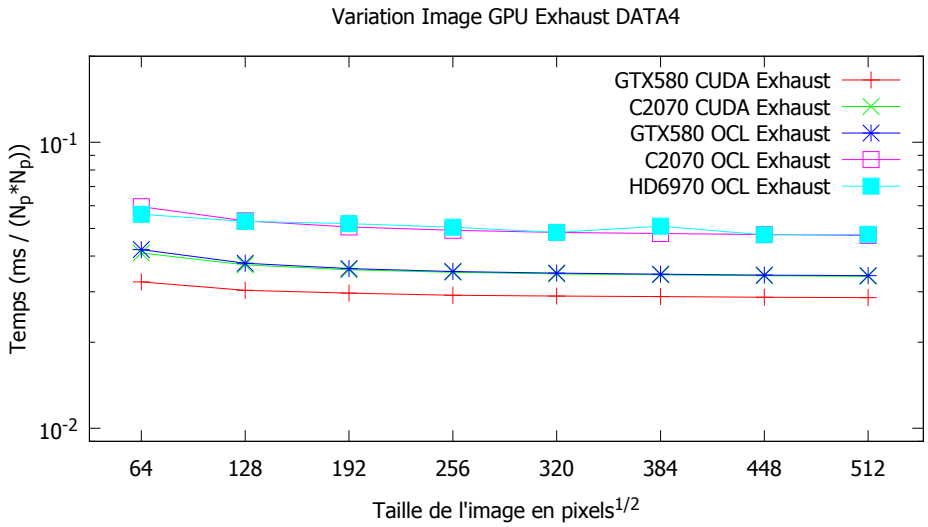


FIGURE 3.19 – Temps de calcul normalisé pour l'implémentation *Exhaust* sur GPU pour le jeu de données DATA 4 et pour des tailles d'images de 64×64 à 512×512.

Concernant l'implémentation *Interp*, les figures 3.20 et 3.21 présentent respectivement les résultats pour les jeux DATA 1 et DATA 4. On peut constater que les comportements entre les deux jeux sont extrêmement différents. En effet, lorsque la résolution augmente, le nombre de tuiles d'interpolation est lui fixe, et c'est le nombre de pixels traité par tuile qui augmente.

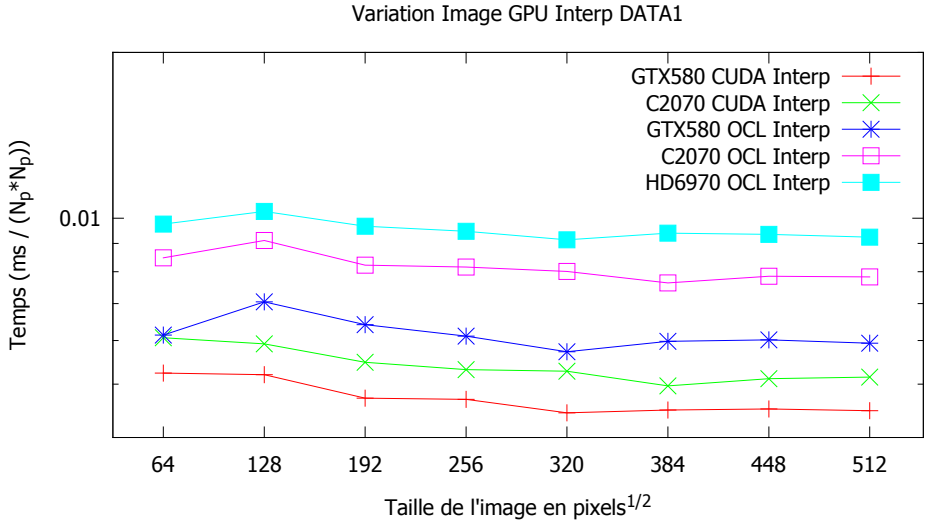


FIGURE 3.20 – Temps de calcul normalisé pour l'implémentation *Interp* sur GPU pour le jeu de données DATA 1 et pour des tailles d'images de 64×64 à 512×512.

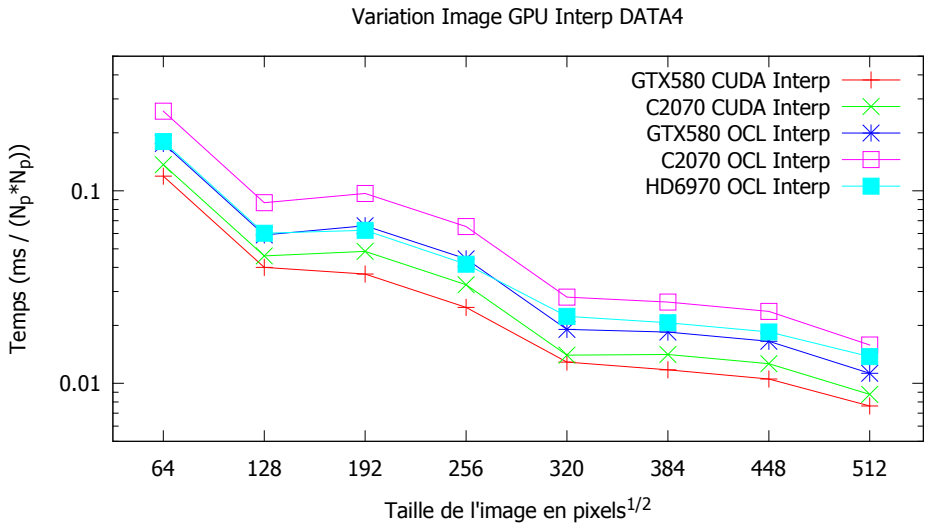


FIGURE 3.21 – Temps de calcul normalisé pour l'implémentation *Interp* sur GPU pour le jeu de données DATA 4 et pour des tailles d'images de 64×64 à 512×512.

Pour le jeu DATA 1, on constate que les courbes sont relativement constantes. Cela s'explique par le fait du peu de calculs sur cette configuration et on peut donc conclure que cette implémentation est *memory-bound* sur ce jeu de données. Cette limitation mémoire se situe directement sur les accès mémoire aux signaux. Quant au jeu de données 4, on constate que plus la résolution augmente, plus le coût de calcul par pixel est faible. Lorsqu'on arrive à une résolution de 512×512, la courbe n'est toujours pas sur un palier. Cela veut dire qu'en contre-partie, le calcul nécessaire aux temps de vol est suffisant pour contre-balancer les accès mémoire aux signaux, qui sont linéairement dépendants du nombre de pixels de l'image, même pour ce type de résolutions.

Pour les implémentations *Exhaust*, le calcul est donc linéairement dépendant du nombre de pixels, tandis que pour les implémentations *Interp*, selon la quantité de calcul, on peut constater que cette plage est décalée vers des résolutions supérieures. De manière générale, les résolutions utilisées par défauts sont dans la plage où le temps de calcul par pixel est constant. Nous allons maintenant faire varier le nombre d'éléments du traducteur afin de voir le liens avec le temps de calcul.

3.5.2.3 Impact du nombre d'éléments du traducteur

De la même manière que pour le benchmark précédent, nous allons faire varier les configurations initiales sur le nombre d'éléments du traducteur. Toujours comme précédemment, les jeux de données DATA 2 et DATA 3 ne seront pas présentés ici car n'apportant rien par rapport aux deux autres. OpenCL sur les cartes Nvidia n'est pas présent non plus du fait des courbes quasi identiques à celles de CUDA.

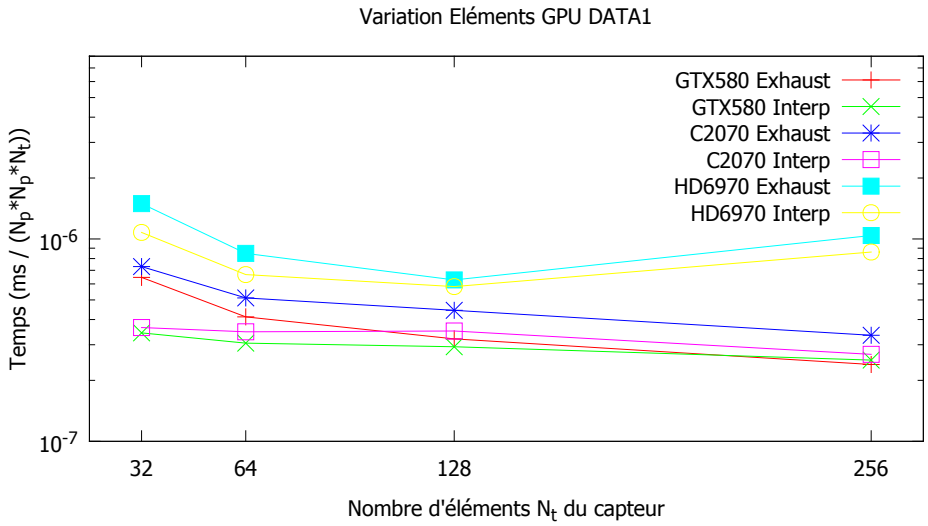


FIGURE 3.22 – Impact du nombre d'éléments du traducteur sur le temps de calcul (normalisés) pour les implémentations *Exhaust* et *Interp* sur GPU et le jeu de données DATA 1.

Les figures 3.22 et 3.22 présentent une variation du nombre d'éléments entre 32 et 256 pour les jeux de données DATA 1 et DATA 4 respectivement. La plage de variation est une plage de valeurs communément utilisées dans des cas réels utilisateur (bien que 256 éléments soient encore peu utilisés du fait du temps de calcul). Les temps affichés sont normalisés en fonction du nombre du pixels de l'image, ainsi que du nombre d'éléments, et ce, afin de vérifier si on est bien linéairement dépendant de ce nombre, comme le montre le tableau de complexité (c.f. tableau 3.1.

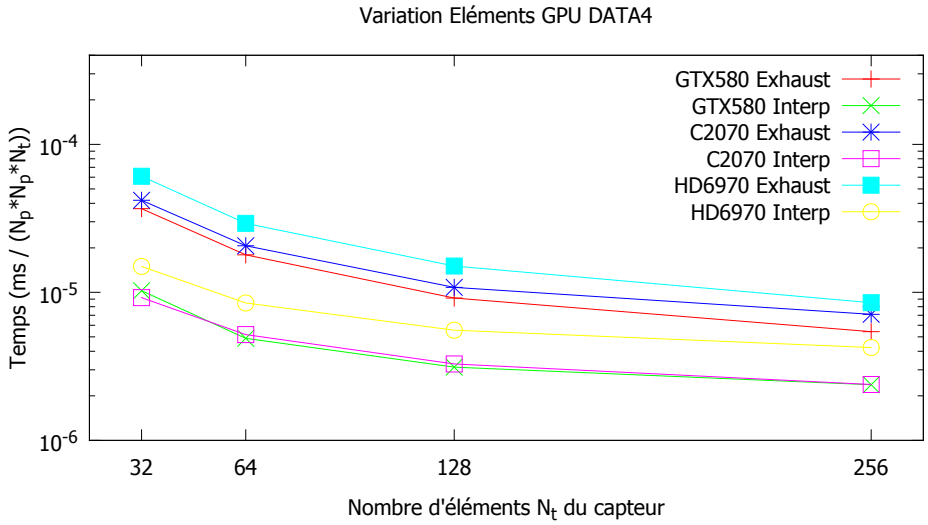


FIGURE 3.23 – Impact du nombre d'éléments du traducteur sur le temps de calcul (normalisés) pour les implémentations *Exhaust* et *Interp* sur GPU et le jeu de données DATA 4.

Pour le jeu de données DATA 1, on constate que les temps de calcul par pixel et par élément sont constants à partir de 64 éléments. Pour de plus petits capteurs, le dimensionnement de la mémoire partagée n'est plus aussi adapté dans l'implémentation qui se veut généraliste. Quant au jeu de données DATA 4, on constate un comportement très différent. En effet, des gains en temps sont observés jusqu'avec 256 éléments, pour les architectures Nvidia. Concernant la HD6970, on constate un coût plus élevé avec 256 éléments sans pouvoir l'expliquer.

Nous avons pu constater que de manière générale, le nombre d'éléments du traducteur à un impact linéaire sur nos GPU. Nous allons maintenant nous intéresser plus particulièrement à la quantité de calcul et à son impact sur nos architectures GPU.

3.5.2.4 Evolution du temps de calcul en fonction du degré du polynôme

Comme nous l'avons vu dans la partie 3.1.3.1, une surface cylindrique implique différents types de calculs selon l'alignement entre l'image, l'orientation du cylindre et le point de reconstruction. Nous avons sélectionné trois configurations différentes, toutes basées sur le jeu de données DATA 2.

1. configuration 1 ou d4 : le traducteur, la génératrice du cylindre et l'image sont sur un même plan. On est donc dans le cas le plus simple ou le polynôme à résoudre est de degré 4,
2. configuration 2 ou d6 : le traducteur est aligné à l'image, mais le cylindre est perpendiculaire, ce qui amène au cas intermédiaire de degré 6,
3. configuration 3 ou d10 : cas général, où les alignements sont différentes des configurations d4 et d6.

La figure 3.24 présente les résultats sur C2070 avec CUDA, pour ces trois configurations différentes, et pour une plage de tailles d'image comme celle présentée précédemment, afin d'observer l'évolution des performances selon la quantité de données à traiter.

Entre les trois configurations, la quantité de calculs augmente d'environ du double à chaque configuration. La figure 3.25 présente une schématisation de la quantité de calculs pour les trois configurations polynomiales. La figure est simplifiée et ne prend en compte que des polynômes à racines réelles. Dans un cas réel comprenant des racines complexes, le nombre de racines complexes est proportionnel au degré du polynôme, ce qui lisse les résultats. Si l'on considère que la quantité de calculs pour la résolution analytique (degré du polynôme inférieur à 3) est à peu près identique à une application de la méthode de Laguerre à un degré donné, on obtient les écarts suivants : $d10 \simeq 2 \times d6 \simeq 4 \times d4$. En pratique, les écarts entre les courbes *Exhaust* de la figure 3.24 sont exactement les mêmes. On a donc un comportement logique.

Concernant l'implémentation *Interp*, on constate que le temps de calcul est identique à partir d'une taille d'environ 300×300 pixels. Cela veut dire que ces configuration sur l'implémentation *Interp* sont limitées par la bande passante mémoire et donc les accès aux amplitudes, alors que l'implémentation *Exhaust* est limitée par la quantité de calculs des temps de vol.

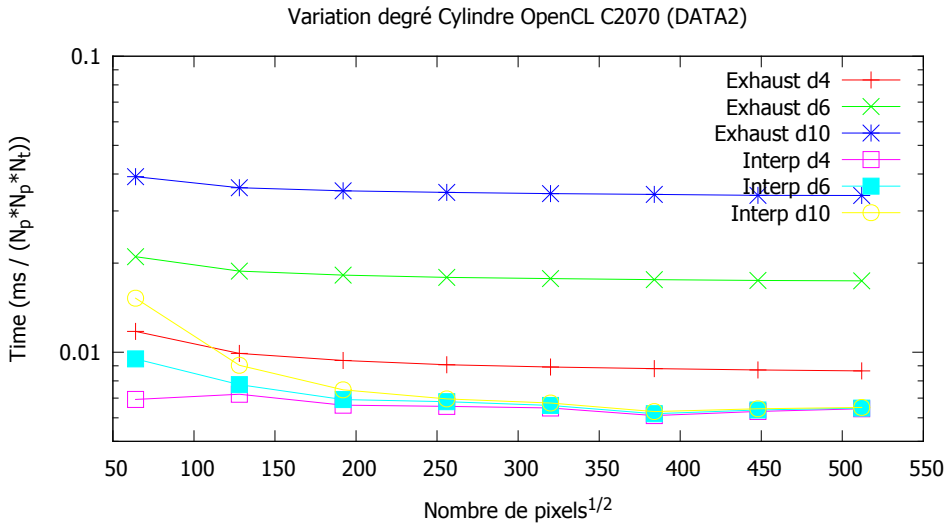


FIGURE 3.24 – Temps de calcul normalisé sur trois configurations impliquant trois quantités de calculs différentes pour des tailles d'images variant entre 64×64 et 512×512.

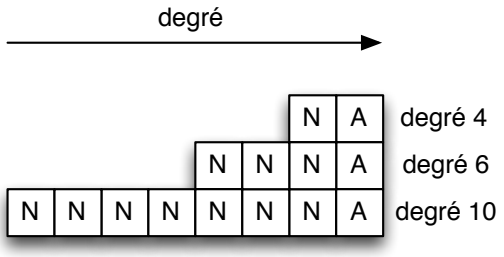


FIGURE 3.25 – Evaluation de la quantité de calculs pour trois configuration cylindriques (A : résolution analytique / N : résolution numérique par la méthode de Laguerre).

3.5.2.5 Coût du calcul sur des pièces à interfaces multiples

Nous allons maintenant nous intéresser plus particulièrement au jeu de données DATA 3, qui est composé de 3 surfaces (pour rappel, deux plans et un cylindre aligné perpendiculairement à la génératrice).

Puisqu'il est nécessaire de tester chaque surface pour effectuer le calcul de temps de vol, nous pourrions nous attendre à des résultats relativement linéaires par rapport au nombre et au type de surfaces dont la pièce est composée. Grâce aux jeux de données DATA 1 et DATA 2, nous pouvons évaluer le coût de calcul des temps de vol en surface plane et cylindrique. Grâce à l'implémentation *Interp*, nous devrions avoir une évaluation du coût des accès mémoire aux signaux. Cette évaluation ne peut être rigoureusement exacte puisque l'entrelacement de calculs et d'accès mémoire recouvre une partie des latences de calculs et d'accès mémoire, mais un ordre de grandeur pourrait être obtenu dans le cas particulier où le calcul des temps de vol en interpolation est considéré comme non significatif. L'opération se résume sous la forme suivante :

$$
\begin{aligned}
T(\text{calcul plan}) &\simeq T_{Exhaust}(DATA1) - T_{Interp}(DATA1) \\
T(\text{calcul cylindre}) &\simeq T_{Exhaust}(DATA2) - T_{Interp}(DATA2) \\
T(\text{accès signaux}) &\simeq T_{Interp}(DATA1) \\
T(\text{DATA 3})_{Exhaust} &\simeq 2 \times T(\text{calcul plan}) + T(\text{calcul cylindre}) + T(\text{accès mémoire})
\end{aligned} \tag{3.15}
$$

Nous avons comparé notre implémentation à nos mesures et nous présentons les résultats pour la GTX 580 avec CUDA dans le tableau 3.7. On constate par rapport à la valeur mesurée pour la configuration DATA 3 que la valeur estimée pour l'implémentation *Exhaust* est extrêmement proche. Quant à l'implémentation *Interp*, la valeur est un peu plus éloignée, mais reste tout de même très proche. Ces mesures se confirment sur la Tesla C2070 et la HD6970 avec des ordres de grandeurs relativement proches.

Tableau 3.7 – Estimation du coût de calcul en cas multi-interfaces.

GTX 580 CUDA	*Exhaust*	*Interp*
DATA 1 (mesuré)	211,8	191,7
DATA 2 (mesuré)	412,3	197,1
T(calcul plan)	20,1	-
T(calcul cylindre)	224,2	-
T(accès mémoire)	191,7	-
DATA 3 (estimé)	456,1	197,1
DATA 3 (mesuré)	458,1	186,2
Erreur relative	2%	6%

Ces résultats ne sont valides que si le calcul par interpolation est limité par la bande passante mémoire, et non par le calcul. En effet, pour des résolutions très basses, ou des jeux de données où le calcul est très largement prépondérant, ce type de calcul ne serait pas juste. En revanche, pour des cas classiques d'utilisation, il est tout à fait possible d'estimer le temps de calcul comme on l'a montré. Nous allons maintenant passer à une estimation du coût de l'utilisation de la double précision dans la mesure où comme nous le savons, les GPU grand public actuels ne disposent pas encore uniquement d'unités flottantes double précision.

3.5.2.6 Impact de la méthode par interpolation des temps de vol

Jusque là, nous n'avons pas comparé directement les deux méthodes de calcul. En effet, l'intérêt de la méthode *Exhaust* peut paraitre limité dans le cas où le calcul par interpolation des temps de vol peut être suffisamment précis et performant. En revanche, cela nous permet de constater que l'interpolation n'est que peu utile dans les cas avec peu de calculs tel qu'un plan simple. Le tableau 3.8 reprend l'ensemble des gains de la méthode par interpolation sur GPU pour les configurations par défaut. On ne constate au maximum qu'un gain de facteur $\times 3{,}3$ pour la C2070 et le jeu de données DATA 4. Evidemment, pour des configurations plus complexes, avec un plus grand nombre de surfaces, l'interpolation sera d'autant plus performante par rapport au calcul exhaustif. Mais on peut constater que pour des configurations de complexité moyenne (comme DATA 3 ou DATA 4), l'écart de performances se situe autour d'un

Tableau 3.8 – Gains par architectures GPU et par API pour l'implémentation *Interp* par rapport à l'implémentation *Exhaust*.

Gains interpolation	CUDA GTX 580	CUDA C2070	OpenCL GTX 580	OpenCL C2070	OpenCL HD6970
DATA 1	$\times$1,1	$\times$1,3	$\times$0,9	$\times$1,0	$\times$1,1
DATA 2	$\times$2,1	$\times$2,4	$\times$1,9	$\times$2,0	$\times$1,7
DATA 3	$\times$2,5	$\times$2,8	$\times$2,2	$\times$2,4	$\times$2,1
DATA 4	$\times$2,7	$\times$3,3	$\times$2,5	$\times$2,4	$\times$2,7

facteur $\times 2$.

Le calcul par interpolation est comme on pouvait s'y attendre plus performant que le calcul exhaustif. En revanche, les performances de l'implémentation exhaustive permettent de constater qu'il devrait être tout à fait possible d'utiliser cette implémentation lorsqu'il est nécessaire d'obtenir une plus grande précision. Ces cas n'ont pas été évalués, mais peuvent se produire avec des pièces très complexes, où les interfaces changent brusquement de type et/ou direction. Dans ces cas, un maillage lâche peut ne pas être assez précis et être source d'artefacts. Il sera donc possible d'avoir avec l'implémentation *Exhaust* de bonnes performances tout en étant plus précis.

3.6 Etude de la parallélisation sur GPP multicoeurs

Nous allons maintenant présenter l'étude de la parallélisation sur GPP multicoeurs. A la différence de l'étude sur GPU, cette section va être découpée en deux parties.

La première partie présente l'analyse d'un benchmark comparable à celui réalisé sur GPU. Les deux implémentations, *Exhaust* et *Interp*, sont présentées. Puis, un ensemble de benchmark est analysé afin d'évaluer le passage à l'échelle des bibliothèques OpenMP et OpenCL dans le cadre de l'algorithme FTP.

La deuxième partie portera sur la réalisation d'une implémentation SIMD de l'algorithme de Laguerre. Les résultats de cette analyse sont présentés indépendamment de ceux réalisés en première partie car l'ensemble des fonctionnalités n'a pas pu être intégré par manque de temps. Nous présenterons tout d'abord un micro-benchmark de l'implémentation de Laguerre en SIMD, puis des résultats de performances de l'implémentation *Exhaust* sur l'algorithme FTP.

3.6.1 Implémentations OpenCL et OpenMP

Sur GPP multicoeurs, nous cherchons une approche qui permette un passage sur les architectures actuelles et celles à venir dans un avenir proche. Ces architectures disposent de 4 à 40 coeurs (4×10 coeurs sur les Sandy-Bridge E), ce qui permet, avec des technologies telles que l'*Hyperthreading*, d'avoir jusqu'à 80 threads de calcul sur une plateforme de type desktop.

Concernant l'implémentation sans interpolation des temps de vol, la parallélisation qui parait la plus intuitive est comme pour le GPU, sur les pixels de l'image comme nous l'avons vu dans la section 3.3.7. Chaque pixel étant indépendant, les threads peuvent se partager leurs calculs de manière optimale. Là où le GPU profite d'une mémoire locale pour stocker les temps de vol, le GPP dispose de ses caches. Sur la plateforme Intel X5690, le cache L1 est de 32 Ko pour les données, ce qui est largement suffisant pour stocker les temps de vol : pour un traducteur de 256 éléments, il est nécessaire de stocker $256\times$ `sizeof(double)` $= 2\,Ko$ par thread.

La méthode par interpolation peut être parallélisée de la même manière que celle sans interpolation. A la place de paralléliser sur les pixels, il est possible de paralléliser sur les tuiles de l'image. En revanche ici, comme nous l'avons dit en introduction de la sous-section, le nombre de tuiles va avoir un rôle important dans la parallélisation. Pour le jeu de données DATA 1, le nombre de tuiles est $\simeq 200$. Cela permet donc d'envisager une parallélisation simple sur les tuiles de la grille d'interpolation pour les architectures actuelles, mais cette quantité de données pourraient être relativement limitant dans un avenir à moyen terme (par rapport à l'augmentation du nombre de coeurs).

Quatre implémentations ont donc été réalisées avec pour cibles les GPP multicoeurs :

- **Les implémentations OpenCL** sont basées sur l'implémentation CUDA qui a été présentée en section 3.5.1. Ce qui diffère ici est qu'on cherche à l'exécuter sur une architecture GPP généraliste au travers des implémentations AMD et Intel. Le dimensionnement des blocs risque d'être donc très différent de ceux du GPU. De plus, comme on l'a vu pour les GPU, l'optimisation architecturale permettant une meilleure utilisation du cache GPU est aussi disponible ici. Le GPP partage effectivement certains niveaux de caches, mais rien ne garantit que ces caches soient efficaces pour des tailles mémoires aussi grandes que celles nécessaires à stocker les temps de vol (pour rappel, pour une configuration standard, $128 \times 8 = 1024$ octets, à la différence des lignes de caches des GPP multicoeurs d'aujourd'hui de 64 octets). Nous allons donc pouvoir voir si les implémentations OpenCL d'AMD et d'Intel arrivent à être efficaces lors de l'utilisation de mémoire partagée de manière intensive.
- **Les implémentations OpenMP** sont basées sur les pseudo-codes présentés en section 15. En revanche pour l'implémentation *Interp*, nous avons inversé l'ordre des boucles de sorte que la boucle interne itère sur les pixels de la tuile. Cela permet d'utiliser le même signal plusieurs fois, et donc de bénéficier des caches du GPP. De plus, on utilise une technique de mémorisation pour les calculs des coefficients d'interpolation qui sont identiques pour chaque tuile. On les calcule donc une unique fois avant de paralléliser et on utilise ce tableau en lecture seule. L'utilisation de l'API OpenMP pour un découpage d'itérations de boucle est relativement immédiat, mais dans la mesure où nous n'étions pas certain de la régularité des calculs, nous avons effectué des tests sur les différents modes d'ordonnancement : *static*, *guided* et *dynamic*) Nous avons constaté des performances très variables selon les implémentations. Nous discuterons de ces variations par la suite et nous fixerons un mode d'ordonnancement pour le reste des benchmarks. Pour les temps de vol, on alloue un espace mémoire privé à chaque thread pour éviter tout conflit. Pour le reste, il s'agit de données partagées en lecture. L'image est, elle-aussi , partagée en lecture, puisque les threads écrivent sur des pixels distincts.

L'implémentation OpenMP *Exhaust* aurait pu faire l'objet d'une optimisation supplémentaire que nous n'avons pas eu le temps d'intégrer. Il s'agit d'une optimisation telle que celle utilisée sur GPU permettant de gagner en localité sur les accès aux signaux. Il serait nécessaire de calculer les temps de vol de plusieurs pixels à la fois, pour ensuite effectuer l'extraction et la sommation des amplitudes sur le même signal pour les différents

3.6.2 Analyse des performances des GPP multicoeurs

Nous allons maintenant nous intéresser aux benchmarks. Nous commencerons par paramétrer nos implémentations OpenCL. Ensuite nous observerons le passage à l'échelle et nous comparerons les performances obtenues via OpenMP à celle obtenues via OpenCL. Nous effectuerons ensuite des benchmarks de variations sur la quantité de pixels et d'éléments du traducteurs ultrasons pour vérifier l'impact de ces deux paramètres, généralement variables pour des reconstructions. Enfin, nous observerons les gains obtenus par la méthode de calcul par interpolation.

3.6.2.1 Paramétrage OpenCL

Comme nous le savons, il est nécessaire d'effectuer un découpage en blocs de threads pour exécuter du code via l'API OpenCL. Le dimensionnement de ce bloc a un réel impact sur les temps de calculs. Et comme nous l'avons vu sur GPU, le nombre de pixels par bloc, que ce soit pour l'implémentation *Exhaust* ou *Interp* est à dimensionner aussi.

Autant sur GPU nous avions une idée du dimensionnement du bloc de threads ainsi que du nombre de pixels par bloc, en se basant sur les limitations hardware du GPU (particulièrement de la mémoire partagée), autant sur GPP, nous n'avons pas ces informations. Nous avons donc balayé un ensemble de tests afin d'obtenir un minimum local pour chaque configuration de reconstruction.

On a donc sélectionné les valeurs suivantes :
- implémentation AMD *Exhaust* : 8 threads par bloc, et 4 pixels traités par bloc,
- implémentation AMD *Interp* : 32 threads par bloc et 8 pixels traités par bloc,
- implémentation Intel *Exhaust* : 8 threads par bloc et 2 pixels traités par bloc,
- implémentation Intel *Interp* : 32 threads par bloc et 8 pixels traités par bloc.

Les valeurs sont donc très proches entre l'implémentation Intel et AMD, ce qui n'a pas toujours été le cas dans nos différentes évaluations d'OpenCL jusqu'ici, ces valeurs ayant été variables entre les versions de SDK et de pilotes.

3.6.2.2 Impact des modes d'ordonnancements d'OpenMP

Lors de nos tests, nous avons fait varier le mode d'ordonnancement d'OpenMP afin d'observer le comportement selon les implémentations. Nous avons sélectionné les valeurs pour DATA1. Ces résultats sont représentatifs des trois autres jeux de données. Le tableau 3.9 présente ces résultats.

On peut tout d'abord constater que les performances sont très variables pour les deux implémentations. Pour *Exhaust*, l'ordonnanceur dynamique est bien plus performant et permet d'obtenir jusqu'à un facteur $\times 14{,}6$ contre seulement 9 en *static*. Cela montre que les calculs sont relativement irréguliers et qu'il est nécessaire de mieux répartir les tâches de calculs. Pour l'implémentation *Interp*, nous n'avons pas réussi à comprendre clairement le résultat. En effet,

Tableau 3.9 – Temps de calcul en ms en mono et multithread, avec et sans *Hyperthreading* pour les implémentations *Exhaust* et *Interp* et les trois variations de l'ordonnanceur OpenMP avec la taille de bloc par défaut (jeu de données DATA 1).

	# threads	1	12 (sans HT)	24 (avec HT)	Gain 12 threads	Gain 24 threads
Exhaust	*dynamic*	10962	869	750	$\times$12,6	$\times$14,6
	static	10962	1118	1225	$\times$9,8	$\times$9,0
	guided	10962	994	1027	$\times$11,0	$\times$10,7
Interp	*dynamic*	4810	784	635	$\times$6,1	$\times$7,6
	static	4810	480	450	$\times$10,0	$\times$10,7
	guided	4810	452	443	$\times$10,6	$\times$10,9

l'ordonnanceur dynamique donne des performances très mauvaises. Cela est probablement dû à des tâches de calcul plus courtes et donc une demande beaucoup plus élevée au thread maitre. Toujours est-il que le mode le plus efficace s'avère être *guided* avec un facteur $\times 10{,}9$ d'accélération dans notre cas.

On peut aussi noter que l'*Hyperthreading* ne change pas la hiérarchie. Il ne semble donc pas qu'il y ait d'effets de caches trop importants, même avec 24 threads. Nous avons donc décidé de fixer le mode dynamique pour l'implémentation *Exhaust* et le mode guidé pour l'implémentation *Interp* dans les résultats qui suivent.

3.6.2.3 Impact de la résolution de l'image

Dans cette partie, nous allons vérifier si le temps de calcul est proportionnel au nombre de pixels traités. Nous avons donc effectué des tests sur différents jeux de données, et sélectionné DATA 1 et DATA 4 comme représentatifs de l'ensemble évalué. Comme pour la variation GPU vue dans la section 3.5.2.2, les courbes pour DATA 1 et DATA 4 ne sont pas à comparer directement car le ratio obtenu entre la maille d'interpolation et la maille pixel n'est pas le même pour ces deux jeux de données.

La figure 3.26 reprend les résultats obtenus pour une variation de taille d'image entre 64×64 et 512×512 sur le jeu de données DATA 1. Les temps de calculs sont normalisés en fonction du nombre de pixels de l'image. On a choisi d'inclure à la fois le calcul sur un seul thread, et la version parallélisée sur 24 afin de vérifier que selon la taille, le passage à l'échelle reste efficace. On constate de manière générale, que pour l'ensemble des implémentations, le temps de calcul par pixel est quasiment constant à partir d'une résolution d'image de 200×200. Nous constatons aussi que l'implémentation la plus performante est OMP24 *Exhaust*, ce qui peut paraitre surprenant. Nous l'expliquerons dans la section consacrée au passage à l'échelle.

La figure 3.27 présente une variation de la résolution pour le jeu de données DATA 4. On constate ici que les implémentations utilisant le calcul par interpolation des temps de vol n'est pas constant sur l'intervalle donné. L'implémentation n'est donc pas limitée par la bande passante, mais bien par le calcul.

Pour l'implémentation *Exhaust*, on est bien linéairement dépendant de pixels de l'image. Quant à l'implémentation *Interp*, le cas se présente à partir d'une certaine résolution. Mais on

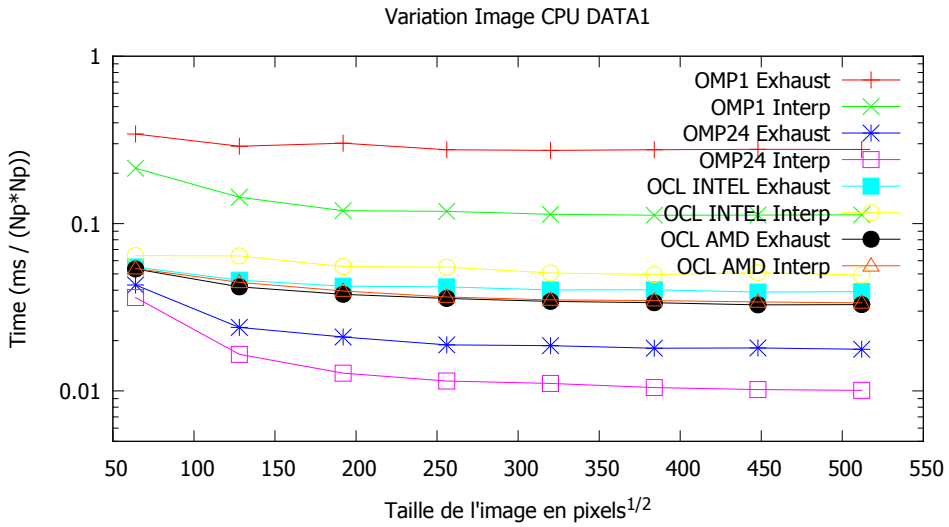


FIGURE 3.26 – Temps de calcul normalisé pour les implémentations *Exhaust* et *Interp* sur GPP pour le jeu de données DATA 1 et pour des tailles d'images de 64×64 à 512×512.

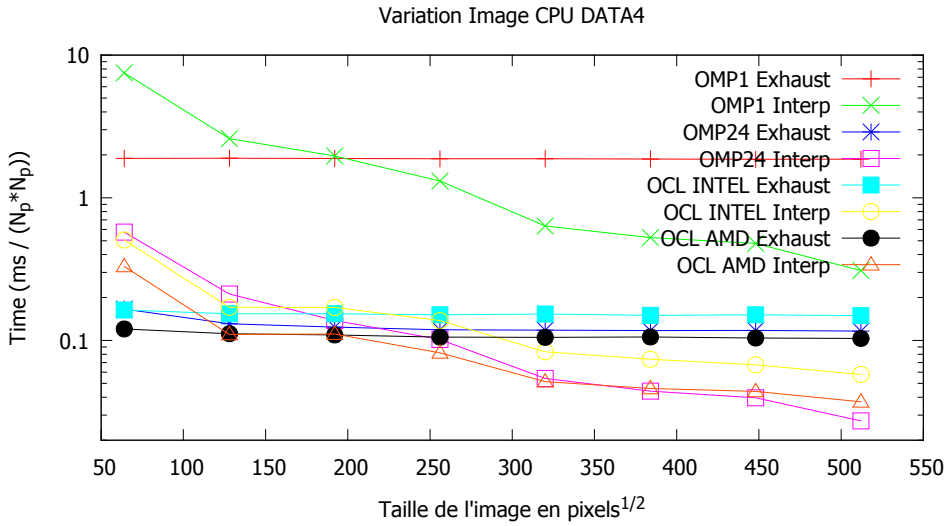


FIGURE 3.27 – Temps de calcul normalisé pour les implémentations *Exhaust* et *Interp* sur GPP pour le jeu de données DATA 4 et pour des tailles d'images de 64×64 à 512×512.

constate bien que pour de petites résolutions, le calcul bien prépondérant. Nous allons maintenant observer le comportement pour différentes tailles de traducteurs.

3.6.2.4 Impact du nombre d'éléments du traducteur

Nous effectuons un benchmark sur le nombre d'éléments du traducteur afin de savoir si l'on est bien linéaire en fonction de ce nombre d'éléments. Nous présentons comme pour le benchmark précédent des résultats pour les jeux de données DATA 1 et DATA 3, qui sont représentatifs de l'ensemble évalué.

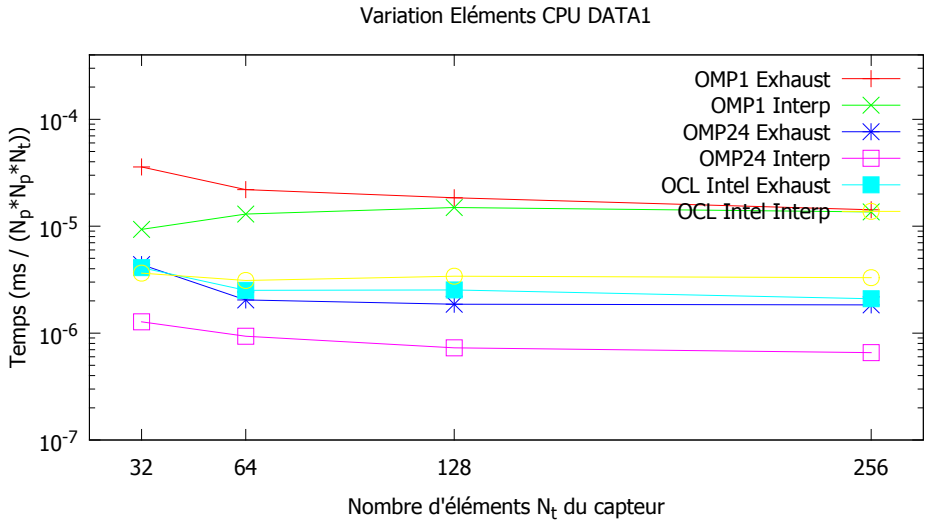


FIGURE 3.28 – Variation du nombre d'éléments du traducteur pour les implémentations *Exhaust* et *Interp* sur GPP et le jeu de données DATA 1.

Le jeu de données DATA 1 donne des temps normalisés très stables, comme on peut le voir sur la figure 3.28 sauf pour des traducteurs 32 éléments où le temps est légèrement plus important pour l'implémentation *Exhaust*. Nous ne présentons pas les résultats pour le jeu de données DATA 4 où la seule différence réside dans le fait que la pente descendante commence dès 32 éléments. Par conséquent, toutes les courbes suivent une même pente.

De manière générale, plus le nombre d'éléments est important plus le temps est réduit comme on a pu le voir sur GPU. On masque certaines latences dues aux calculs de temps de vol et aux accès mémoires. Nous allons maintenant nous intéresser au passage à l'échelle des GPP multicoeurs.

3.6.2.5 Passage à l'échelle OpenMP

Nous avons synthétisé les résultats obtenus pour nos différents jeux de données dans les tableaux 3.10 et 3.11, respectivement pour les implémentations *Exhaust* et *Interp*. Nous avons inclus trois temps de calculs par jeu de données :

- monothread,
- 12 threads, *Hyperthreading* désactivé,
- 24 threads, *Hyperthreading* actif.

Pour l'implémentation *Exhaust*, on peut constater des gains excellents avec *Hyperthreading* avec un facteur d'accélération compris entre $\times 15{,}2$ et $\times 16{,}6$ selon les jeux de données. On a donc un gain surlinéaire de plus de 30% grâce à *Hyperthreading*. Sans *Hyperthreading*, les résultats sont assez stables et légèrement surlinéaires ce qui s'explique par l'utilisation de tous les caches L1 et L2.

Nombre de threads	1	12 (sans HT)	24 (avec HT)	Gain 12 threads	Gain 24 threads
DATA 1	$2{,}14 \cdot 10^{-3}$	$1{,}68 \cdot 10^{-4}$	$1{,}41 \cdot 10^{-4}$	$\times 12{,}8$	$\times 15{,}2$
DATA 2	$5{,}33 \cdot 10^{-3}$	$4{,}16 \cdot 10^{-4}$	$3{,}32 \cdot 10^{-4}$	$\times 12{,}8$	$\times 16{,}1$
DATA 3	$5{,}45 \cdot 10^{-3}$	$4{,}34 \cdot 10^{-4}$	$3{,}33 \cdot 10^{-4}$	$\times 12{,}6$	$\times 16{,}4$
DATA 4	$7{,}68 \cdot 10^{-2}$	$6{,}39 \cdot 10^{-3}$	$4{,}62 \cdot 10^{-3}$	$\times 12{,}0$	$\times 16{,}6$

Tableau 3.10 – Temps de calcul normalisé ($ms/N_p/N_e$) GPP, avec et sans *Hyperthreading*, et passage à l'échelle pour l'implémentation *Exhaust*

Concernant maintenant l'implémentation *Interp*, on constate que les gains sont largement plus faibles pour les jeux de données avec peu de calculs. Avec ou sans *Hyperthreading*, les gains sont de l'ordre d'un facteur $\times 11$ pour les trois premiers jeux de données, soit quasiment 50% inférieurs à ceux de l'implémentation *Exhaust*. Cela s'explique par l'intensité arithmétique plus faible que l'implémentation *Exhaust* ce qui implique une pression plus importante sur les caches. On constate d'ailleurs que pour DATA 4, les facteurs d'accélérations sont à nouveau excellents atteignant un facteur $\times 15{,}1$. Cela montre bien que pour la configuration par défaut pour DATA 4, l'implémentation est limitée par le calcul, tandis que pour les trois autres jeux de données, elle est limitée par la bande passante et les accès aux caches.

Tableau 3.11 – Temps de calcul normalisé ($ms/N_p/N_e$) GPP et passage à l'échelle pour l'implémentation *Interp*

Nombre de threads	1	12 (HT OFF)	24 (HT ON)	Gain 12 threads	Gain 24 threads
DATA 1	$1{,}87 \cdot 10^{-3}$	$8{,}86 \cdot 10^{-5}$	$8{,}71 \cdot 10^{-5}$	$\times 10{,}9$	$\times 11{,}0$
DATA 2	$2{,}07 \cdot 10^{-3}$	$9{,}41 \cdot 10^{-5}$	$9{,}25 \cdot 10^{-5}$	$\times 11{,}3$	$\times 11{,}5$
DATA 3	$2{,}08 \cdot 10^{-3}$	$9{,}71 \cdot 10^{-5}$	$8{,}99 \cdot 10^{-5}$	$\times 10{,}7$	$\times 11{,}5$
DATA 4	$1{,}96 \cdot 10^{-2}$	$1{,}45 \cdot 10^{-3}$	$1{,}11 \cdot 10^{-3}$	$\times 11{,}7$	$\times 15{,}1$

3.6.2.6 Comparaison entre OpenMP et OpenCL

Intéressons nous maintenant aux différences entre OpenMP et OpenCL. Le tableau 3.12 récapitule les facteurs d'accélérations pour les différents jeux de données en OpenMP avec 24 threads et les deux implémentations OpenCL, Intel et AMD.

Tableau 3.12 – Comparaison des accélérations entre OpenMP et OpenCL Intel/AMD (avec 24 threads) par rapport aux implémentations *Exhaust* et *Interp* monothreadées.

		temps (ms)	OpenMP 24th	OpenCL Intel	OpenCL AMD
Exhaust	DATA 1	10,97	$\times$15,2	$\times$6,6	$\times$7,6
	DATA 2	27,28	$\times$16,1	$\times$10,5	$\times$13,1
	DATA 3	27,91	$\times$16,4	$\times$10,1	$\times$12,8
	DATA 4	393,34	$\times$16,6	$\times$12,8	$\times$18,3
Interp	DATA 1	4,93	$\times$11,0	$\times$2,2	$\times$3,1
	DATA 2	5,43	$\times$11,5	$\times$2,4	$\times$3,3
	DATA 3	5,31	$\times$11,5	$\times$2,4	$\times$3,3
	DATA 4	86,55	$\times$15,2	$\times$6,2	$\times$10,0

Concernant l'implémentation *Exhaust*, les accélérations OpenMP sont très stables tandis qu'en OpenCL, pour le jeu DATA 1, ces accélérations sont faibles avec un facteur $\times$6,6 pour Intel, et 7,6 pour AMD. Ces facteurs remontent ensuite pour les jeux de données avec plus de calculs. Cela pourrait s'expliquer par le fait que l'ordonnanceur OpenCL est moins efficace que celui d'OpenMP et que plus on a de calculs, plus les latences de gestion de threads sont recouvertes, jusqu'à les masquer complètement. Cela se vérifie avec le jeu DATA 4, où AMD parvient à dépasser en performances OpenMP avec un facteur $\times$18,3 par rapport à l'implémentation séquentielle. Même si la structure du code OpenCL diffère de celle d'OpenMP, il n'est pas évident d'expliquer ce résultat. Une hypothèse serait que le code est vectorisé à un certain niveau, mais nous ne l'avons pas vérifié.

Si maintenant on observe les résultats obtenus sur l'implémentation *Interp*, OpenCL se trouve ici en grande difficulté par rapport à OpenMP. OpenCL Intel ne parvient pas à dépasser un facteur $\times$7 d'accélération sur le jeu DATA 4 et est à peine deux fois plus rapide que l'implémentation séquentielle pour les trois autres jeux de données. Cela pourrait s'expliquer une nouvelle fois du fait de l'ordonnanceur, qui diffère entre l'implémentation AMD et Intel (en l'occurrence, l'ordonnanceur de TBB, puisqu'Intel utilise TBB comme base pour OpenCL). Même si AMD semble plus efficace, les gains sont faibles lorsqu'il n'y a que peu de calculs, et atteignent jusqu'à un facteur $\times$10 sur le jeu de données DATA 4.

OpenMP dispose d'une avance certaine sur OpenCL. Les résultats sont - sauf exception - bien meilleurs que ceux d'OpenCL. On notera les performances de l'implémentation Intel, bien en deçà de celles d'AMD. Il semblerait qu'il y ait encore une certaine marge pour améliorer le fonctionnement des ordonnanceurs et plus généralement des implémentations multicoeurs.

3.6.3 Implémentation SIMD de l'algorithme de Laguerre

Etant donné les résultats de précision obtenus en section 3.4.3, nous nous sommes interrogés sur la possibilité de réaliser une implémentation SIMD de l'algorithme de Laguerre et des performances que nous pouvions en obtenir. En effet, la simple précision pouvant être utilisée, sous réserve de conditions relativement peu contraignantes, nous avons donc réalisé une implémentation SSE de l'algorithme de Laguerre.

Afin de passer un algorithme en SIMD, il est bien souvent nécessaire de le transformer de différentes façons afin de pouvoir utiliser les instructions SIMD. Il se pose alors le problème de savoir ce qu'il est nécessaire d'ajouter en plus comme instructions de contrôle. Nous allons voir en première partie comment nous avons adapté l'algorithme de Laguerre. Ensuite, nous effectuerons une analyse de l'implémentation avec un micro-benchmark. Enfin, nous effectuerons une comparaison avec les implémentations non-vectorisées.

3.6.3.1 Présentation de l'implémentation SIMD de l'algorithme de Laguerre

Afin de réaliser l'implémentation de l'algorithme de Laguerre en SIMD, il faut pouvoir résoudre principalement deux problèmes : la gestion du stockage des coefficients en mémoire et la gestion du nombre d'itérations des algorithmes de Horner et Laguerre.

Soient deux polynômes $P1$ et $P2$, de degré n et m, $n < m$:

$$P1(X) = a_0 + a_1X + a_2X^2 + ... + a_nX^n \tag{3.16}$$
$$P2(X) = b_0 + b_1X + b_2X^2 + ... + b_mX^m \tag{3.17}$$

Si l'algorithme de Laguerre a une forte intensité arithmétique, ce n'est pas le cas de l'algorithme de Horner. Pour rappel, il permet d'écrire P1 sous la forme suivante :

$$P1(X) = a_0 + X(a_1 + X(a_2 + ... + X(a_{n-1} + X(a_n)))) \tag{3.18}$$

Ce dernier dispose d'une addition et d'une multiplication par accès mémoire. L'intensité arithmétique s'améliore dans le cas où la racine évaluée est complexe, mais reste cependant faible. Il est donc souhaitable d'une part, de stocker les coefficients du polynôme et de ses différentes factorisations dans la même zone mémoire et d'autre part de réaliser l'évaluation des dérivées première et seconde à la volée. Cette évaluation est réalisée à l'aide du schéma de calcul basé sur la représentation de Horner. Nous appellerons *Horner3* l'algorithme permettant d'évaluer en plus les deux dérivées du polynôme.

Les coefficients des polynômes sont stockés dans l'ordre décroissant plutôt que croissant. Ainsi, le coefficient a_n sera stocké en `P1[0]`. De plus, afin de pouvoir gérer l'évaluation de plusieurs polynômes de différents degrés, le coefficient b_m de $P2$ sera aussi stocké dans la case mémoire zéro `P2[0]`. Cette implémentation est très utilisée sur des puces telles que les DSP car elle permet de mettre en évidence les opérations de multiplication et addition ou MAC qui sont souvent des opérations réalisées en 1 cycle.

Les algorithmes 6 et 7 présentent les pseudo-codes des algorithmes d'évaluation de *Horner* et de *Horner3* pour un polynôme P de degré n dont on évalue la valeur pour x.

Algorithme 6: Evaluation de la valeur d'un polynôme par la méthode de Horner.

entrée : polynôme P de degré n
entrée : valeur x évaluée
sortie : valeur f du polynôme

```
// Initialisation
```
1 f = $P[0]$
2 **for** $k \leftarrow 1$ **to** n **do**
3 f = $x\times$ f + $P[k]$

Algorithme 7: Evaluation de la valeur d'un polynôme et de ses deux premières dérivées par la méthode de *Horner3*.

entrée : polynôme P de degré n
entrée : valeur x évaluée
sortie : valeur f du polynôme
sortie : valeur fp de la dérivée du polynôme
sortie : valeur fpp de la dérivée seconde du polynôme

```
// Initialisation
```
1 f = 0
2 fp = 0
3 fpp = 0
4 **for** $k \leftarrow 0$ **to** $n-2$ **do**
5 f = $x\times$f + $P[k]$
6 fp = $x\times$fp + f
7 fpp = $x\times$fpp + fp
```
// Fin des calculs
```
8 f = $x\times$ f + $P[n-1]$
9 fp = $x\times$ fp+ f
10 f = $x\times$ f + $P[n]$
11 fpp = $2\times$fpp

Le nombre de tours de boucle de l'algorithme de Horner est égal à la valeur du plus grand degré. Il est donc nécessaire de neutraliser l'évaluation pour les autres polynômes. Prenons le cas ou $m = n - 1$:

$$P1_i = P1_{i+1} \times X + b_i \tag{3.19}$$
$$S1_i = P1_{i+1} \times P1_i \tag{3.20}$$

L'évaluation de $P2$ donne pour une boucle allant de n à zéro :

$$P2_0 = P2_1 \times X + b_0 \tag{3.21}$$
$$S2_{-1} = P2_0 \times X + b_{-1} \tag{3.22}$$

L'addition de b_{-1} est inutile et fausse mais elle n'est pas problématique, car lors de la factorisation du polynôme, la case `P2[n]` associée à b_{-1} contient zéro, élément neutre de l'addition. P_{-1} vaut donc $P_0 \times x$. C'est donc la dernière multiplication par x qui pose problème et qui doit être neutralisée. Dans le cas général, le polynôme qui est évalué est en fait $P2(x) \times x^{n-m}$. L'algorithme de Laguerre trouvera donc zéro comme racine, éventuellement multiple.

L'algorithme de Laguerre pose le même problème du fait de son aspect itératif. Il y a deux conditions d'arrêt pour Laguerre : en début de boucle et en fin de boucle. La condition de sortie

en fin de boucle correspond au cas où le critère de convergence est atteint. Celle en début de boucle correspond soit au cas où la dérivée première est nulle, pour éviter une division par zéro, soit au critère de Adams. Il faut donc mémoriser la dernière valeur correcte de x à travers les itérations successives. Toutes ces opérations de neutralisation/mémorisation sont réalisées via des instructions de masquage et/ou de sélection. Nous appelons *HornerEQ* l'implémentation de test où tous les polynômes sont de même degré et *HornerNEQ* le cas réel avec la gestion de polynômes de degrés différents. Nous allons utiliser l'implémentation *HornerEQ* afin d'observer le coût des instructions de contrôle.

3.6.3.2 Micro-benchmark de Laguerre

Afin d'évaluer l'impact du SIMD pour FTP, nous avons réalisé un micro-benchmark de Laguerre et de Horner pour bien comprendre leur comportement. Nous avons réalisé les comparaisons suivantes :

- Horner réel vs complexe : coût de l'évaluation d'un polynôme réel pour une valeur réelle ou complexe
- *HornerEQ* vs *HornerNEQ* : coût de l'ajout des instructions de contrôle permettant de traiter des polynômes de degrés différents (ce qui est nécessaire).
- *Horner3* vs $3\times$ *Horner* : estimation du gain de l'évaluation des dérivées à la volée plutôt que de les calculer une à une.
- Laguerre complet : comparaison de l'implémentation SIMD de Laguerre.

L'évaluation de l'algorithme de Laguerre est bien plus complexe. Le nombre d'itérations pour trouver une racine dépend bien sûr des polynômes et de la valeur initiale fournie à l'algorithme. Le nombre d'appels à la fonction de Laguerre va dépendre des racines du polynôme. Si celui-ci n'a que des racines réelles, la décroissance du degré ira de 1 en 1, tandis que s'il n'y a que des racines complexe, elle ira de 2 en 2.

Les benchmarks sont réalisés sur un Intel Xeon Westmere 2x4 coeurs cadencé à 2,66 GHz. Les mesures sont réalisés en cycles à l'aide de la primitive `__rdtsc()`, instruction disponible sur les processeurs Intel et qui permet d'effectuer des mesures aux cycle près. L'utilisation de la bibliothèque OpenMP est effectuée pour confirmer le passage à l'échelle et l'Hyperthreading est actif. Nous avons utilisé ICC 12.1 avec l'option `-no-vec` et effectué une comparaison rapide avec GCC 4.2 (compilateur par défaut sur la machine de benchmark), ce qui nous a montré un écart de performances relativement stable d'environ 10% en faveur d'ICC.

Horner réel vs complexe

Nous commençons par l'évaluation de la valeur du polynôme à l'aide de la méthode de Horner. Pour les besoins de l'implémentation SIMD, nous avons supprimé le branchement entre réel et complexe. Avant cela, nous avons évalué ce coût en mesurant une implémentation SIMD de Horner pour des racines réelles et l'implémentation que nous allons utiliser dans Laguerre, en complexe cette fois. Le tableau 3.13 présente ces mesures pour l'implémentation SSE F32. Nous avons retiré délibérément les valeurs de degrés intermédiaires du fait de la linéarité des temps de calculs en fonction du degré.

On peut observer que le coût de la version complexe est inférieur à 2 fois le coût de la version réelle. Dans le cas réel, on a une addition et une multiplication par itération sur le degré, tandis qu'en complexe, on a quatre additions et une multiplication. On devrait donc avoir un facteur 2 étant donné les latences de chaque opération. Nous n'avons pas d'explication claire, mais nous pensons qu'il peut s'agir d'une différence au niveau du pipelining des opérations. Il

Tableau 3.13 – Micro-benchmark de Horner sur l'algorithme de Laguerre en SSE F32

Type	Degré	Cycles	Ratio C/R	Ratio 16/4
Réel	4	35	-	-
Complexe	4	46	$\times$1,31	-
Réel	16	97	-	$\times$2,77
Complexe	16	175	$\times$1,80	$\times$3,80

serait intéressant d'effectuer des tests avec d'autres compilateurs et d'aller voir dans l'assembleur, mais nous n'avons pas eu le temps de le faire. L'écart entre réel et complexe atteint 1,80 en degré 16, ce qui est plus logique. On a donc un overhead présent pour le degré 1 ce qui explique que l'on observe pas directement la linéarité dans les résultats. Dans le cas complexe, on observe bien un facteur 3,8 qui tend vers 4 comme nous pouvions l'attendre.

Horner3* vs *3x Horner

Nous utilisons donc maintenant uniquement le calcul complexe, même pour l'évaluation de racines réelles. Nous allons maintenant mesurer les performances de l'évaluation de Horner des dérivées première et seconde par rapport à une implémentation qui les évalue indépendamment. Nous appelons donc *Horner3* l'implémentation à la volée et *3x Horner* l'implémentation indépendante. Le tableau 3.14 présente les performances obtenues.

Tableau 3.14 – Micro-benchmark de *Horner3* vs *3x Horner*

Degré	$3 \times Horner$ (cycles)	*Horner3* (cycles)	Gain *Horner3*
4	101	105	$\times$0,96
16	541	379	$\times$1,42

On peut observer qu'en degré 4, le calcul à la volée n'est pas plus intéressant. Cette valeur est relativement faible, mais on peut observer, pour des degrés supérieurs, des gains de plus en plus importants, avec un gain de 42% en degré 16. Il est donc tout à fait pertinent d'utiliser l'implémentation à la volée.

HornerEQ* vs *HornerNEQ

Dans le cadre de l'algorithme de Laguerre, il est tout à fait probable que deux polynômes soient de degré différent. Nous avons donc effectué des mesures pour évaluer ce coût. Le tableau 3.15 les présente pour l'implémentation SSE F32 ; les mesures effectuées en F64 sont redondantes et nous ne les présentons pas ici. *HornerEQ* représente le cas où les polynômes sont forcément de même degré et *HornerNEQ* le cas général où l'on a ajouté les instructions de contrôle nécessaires. Nous avons ajouté une autre implémentation de Horner, où l'on effectue le calcul du critère de Adams. Cela ajoute, en opérations réelles, une valeur absolue, une addition et une multiplication. Le surcoût est donc relativement faible. Nous appelons cette implémentation *Horner4*.

Que l'on observe l'implémentation *Horner3* ou *Horner4*, on peut observer un comportement relativement identique. Le coût des instructions de contrôle est compris en 30% et 40% en degré 4. Ce coût augmente en fonction du degré pour arriver à 70% supplémentaires en degré 16.

Tableau 3.15 – Micro-benchmark de *HornerEQ* vs *HornerNEQ*

Méthode	Degré	Cycles	Ratio NEQ/EQ	Ratio 16/4
Horner3 EQ	4	73	-	-
Horner3 NEQ	4	105	$\times 1{,}43$	-
Horner3 EQ	16	220	-	$\times 3{,}01$
Horner3 NEQ	16	379	$\times 1{,}72$	$\times 3{,}61$
Horner4 EQ	4	83	-	-
Horner4 NEQ	4	122	$\times 1{,}46$	-
Horner4 EQ	16	254	-	$\times 3{,}06$
Horner4 NEQ	16	425	$\times 1{,}67$	$\times 3{,}48$

Il faut bien comprendre que c'est un coût nécessaire et que nous n'avons pas le choix de faire autrement. De plus ce coût peut être augmenté dans le cas où un polynôme aurait terminé plus tôt que l'autre (ou l'un des autres) calculés de manière vectorielle.

Si l'on observe les performances en fonction du degré, on peut constater un comportement identique à précédemment, avec une évaluation qui s'approche du facteur 4 correspondant à l'écart de quantité de calcul entre degré 4 et 16. Cependant, dans le cas EQ, ce coût est inférieur.

Si maintenant on observe les différences de performances entre *Horner3* et *Horner4*, on peut constater que le coût du critère de Adams diminue en fonction du degré et est compris en 15% et 25%, ce qui est relativement faible.

Le coût par degré pour l'implémentation de Horner avec critère de Adams arrive à 25 cycles par degré et ce, pour calculer la valeur du polynôme, de ses deux dérivées et le critère de Adams.

Microbenchmark de Laguerre

Le tableau 3.16 présente une synthèse des benchmarks réalisés sur l'implémentation de Laguerre. Afin d'effectuer une mesure stable, le nombre d'itérations de Laguerre est fixé à 4. Le cas R est pour un polynôme où toutes les racines sont réelles, et le cas C est le cas où toutes les racines sont complexes. Nous présentons ici uniquement les cas de polynômes de degré 4 et 16 mais nous avons, comme pour tous les benchmarks précédents, effectué les cas intermédiaires pour vérifier que le passage à l'échelle s'effectuait correctement.

Il faut noter que dans le cas scalaire, seul 1 polynôme est résolu par Laguerre, tandis qu'en SSE, ce sont respectivement 2 et 4 polynômes qui sont traités en F64 et en F32. Le gain de performance total est donc à normaliser en fonction du nombre de polynômes traités.

Tableau 3.16 – Micro-benchmark de Laguerre

		Degré	C (cycles)	R (cycles)	Ratio 16/4	Ratio C/R	Ratio F64/F32	Ratio SSE/Scalar
Scalar	F32	4	1299	2441	-	$\times 1{,}88$	-	-
Scalar	F32	16	8976	17675	$\times 6{,}91$	$\times 1{,}97$	-	-
Scalar	F64	4	1378	2592	-	$\times 1{,}88$	$\times 1{,}06$	-
Scalar	F64	16	9426	18635	$\times 6{,}84$	$\times 1{,}98$	$\times 1{,}05$	-
SSE	F32	4	1609	3039	-	$\times 1{,}89$	-	$\times 1{,}24$
SSE	F32	16	11240	21552	$\times 6{,}99$	$\times 1{,}92$	-	$\times 1{,}25$
SSE	F64	4	1724	3268	-	$\times 1{,}90$	$\times 1{,}07$	$\times 1{,}25$
SSE	F64	16	11600	22560	$\times 6{,}73$	$\times 1{,}94$	$\times 1{,}03$	$\times 1{,}23$

On peut d'ailleurs constater qu'à ce niveau, entre degré 4 et 12, on observe un facteur 7 d'écart entre ces deux cas. Cela s'explique par le fait que lorsque l'on se trouve en degré 2, on effectue analytiquement la résolution de manière à éviter une exécution de Laguerre beaucoup plus couteuse. Dans le cas complexe, on revient donc à un appel à Laguerre pour le degré 4 à 7 appels pour le degré 16. Plus évident, le ratio entre R et C se situe autour de 2, ce qui s'explique simplement par le fait qu'il est nécessaire d'effectuer 2 fois plus d'appels à Laguerre pour le cas R.

Concernant le SSE, on observe des performances relativement proches entre F32 et F64. Dans le cas monothreadé présenté dans le tableau, on observe que le coût est moindre en F32, ce qui peut s'expliquer par les différences de latences existantes entre les intrinsèques SSE et SS2 (`__mm_mul_ps` et `__mm_mul_pd` ont respectivement une latence de 4 et 5 cycles avec la microarchitecture Nehalem).

Enfin, si l'on observe le ratio entre SSE et Scalar, on observe un surcoût d'environ 25% qui est dû aux instructions de contrôle nécessaires à la gestion de différents degrés de polynôme et aux variations de nombre d'itérations (correspond aux implémentations notées NEQ précédemment).

En utilisant OpenMP, nous avons mesuré les gains cumulés entre parallélisation et vectorisation et nous les avons regroupés dans le tableau 3.17. Les gains obtenus sont proches des valeurs théoriques avec un facteur 32 (8 coeurs et 4 valeurs par registre SIMD) pour le F32, et 16 (8 coeurs et 2 valeurs par registre SIMD) pour le F64. Les gains obtenus sont mêmes parfois supérieurs, du fait de l'Hyperthreading. En effet, nous ne présentons pas les valeurs pour rester le plus synthétique possible, mais l'Hyperthreading permet d'obtenir des gains surlinéaires avec des gains légèrement supérieurs à un facteur 10 pour 16 threads.

Tableau 3.17 – Gain apporté par OpenMP et le SIMD (SSE) pour l'algorithme de Laguerre dans le cadre du micro-benchmark de Horner.

Gain total	Degré	C	R
F32	4	×28,39	×32,44
F32	16	×29.60	×31,08
F64	4	×16,50	×16,89
F64	16	×15,71	×16,20

3.6.3.3 Performances SIMD dans FTP

Le code SIMD présenté ci-dessus a été intégré dans notre code FTP. Cependant, pour des raisons de temps, nous n'avons pu compléter toutes les implémentations nécessaires à une comparaison directe aux implémentations scalaires. Nous avons donc uniquement intégré le code SIMD dans l'implémentation *Exhaust* et uniquement pour des pièces à surface unique. Nous allons donc présenter des benchmarks portant uniquement sur des reconstructions d'une pièce plane et d'une pièce cylindrique.

Ayant validé les performances de la parallélisation de l'algorithme FTP ainsi que celles de l'implémentation SIMD sur un processeur Nehalem, nous avons jugé qu'il serait intéressant d'observer le comportement sur une machine AMD disposant d'un nombre de coeurs plus important. Nous avons donc effectué nos benchmarks sur une machine AMD disposant de quatre processeurs Opteron 6284 SE, de 2x8 coeurs chacun, faisant un total de 64 coeurs.

Le tableau 3.18 présente les performances obtenues pour l'exécution de FTP sur deux jeux de données : un cas plan, et un cas cylindrique. Les jeux de données DATA1 pour le cas plan et DATA2 pour le cas cylindrique sont utilisés. En revanche, nous n'avons pas pu réutiliser une CAO2D dans la mesure où nous n'avons pas eu le temps de mettre le code à niveau : nous n'avons donc pas pu tester les jeux de données DATA3 et DATA4. Le compilateur utilisé est ICC 13.1 et l'environnement de benchmark est sous Linux.

Tableau 3.18 – Performances des implémentations scalaires et SIMD FTP sur une machine 64 coeurs AMD.

DATA1 (ms)		Mono	Multi (64)	Gain OpenMP
Scalar	F32	26 125	699	×37,3
SSE	F32	9 094	283	×32,1
Scalar	F64	27 872	881	×31,6
SSE	F64	10 733	338	×31,8

DATA2 (ms)		Mono	Multi (64)	Gain OpenMP
Scalar	F32	49 567	1 310	×37,8
SSE	F32	12 229	499	×24,6
Scalar	F64	98 630	2 515	×39,2
SSE	F64	33 166	914	×36,3

Les gains observés sont loins d'être ceux que nous attendions initialement. En effet, le passage à l'échelle au delà de 32 thread est extrêmement peu efficace et le meilleur facteur obtenu n'atteint qu'un $\times 39{,}2$ pour DATA2 dans le cas scalaire F64. Dans le cas cylindrique SIMD F32, le gain est très faible et n'atteint qu'un facteur $\times 24{,}6$. Ce dernier nous semble difficile à expliquer. Cependant, ce problème de passage à l'échelle sur l'architecture Bulldozer semble être une problématique connue comme expliqué ici [Frey, 2012]. Selon le compilateur et les options de compilation, le problème de passage à l'échelle au delà de 32 threads peut apparaitre. Il serait donc nécessaire de tester d'autres compilateurs pour valider cette hypothèse. Parmi les tests à réaliser, il serait intéressant de réaliser une implémentation AVX car les coeurs de l'architecture Bulldozer peuvent soit exploiter deux flux SSE, soit utiliser la FPU partagée entre les deux coeurs pour exécuter un flux AVX. Il est tout à fait possible que le facteur 2 manquant vienne d'une mauvaise exploitation des instructions SSE.

Pour illustrer les différents problèmes rencontrés dans ces benchmarks, voici une sélection de ratios comprenant les gains apportés par le SSE, les gains apportés par le F32 par rapport au F64 et le gain total, par jeu de données. Le tableau 3.19 présente ces ratios.

Tableau 3.19 – Comparaisons de performances entres différentes valeurs issues du tableau 3.18.

Plan (ratios)		Gain SSE Mono	Gain SSE 64 threads	Gain F32 Mono	Gain F32 64 threads	Gain Total
Scalar	F32	-	-	-	-	-
SSE	F32	$\times 2{,}9$	$\times 2{,}5$	-	-	$\times 92{,}3$
Scalar	F64	-	-	$\times 1{,}1$	$\times 1{,}3$	-
SSE	F64	$\times 2{,}6$	$\times 2{,}6$	$\times 1{,}2$	$\times 1{,}2$	$\times 82{,}6$

Cylindre(ratios)		Gain SSE Mono	Gain SSE 64 threads	Gain F32 Mono	Gain F32 64 threads	Gain Total
Scalar	F32	-	-	-	-	-
SSE	F32	$\times 4{,}0$	$\times 2{,}6$	-	-	$\times 99{,}2$
Scalar	F64	-	-	$\times 2{,}0$	$\times 1{,}9$	-
SSE	F64	$\times 3{,}0$	$\times 2{,}8$	$\times 2{,}7$	$\times 1{,}8$	$\times 107{,}9$

Commençons par observer les gains obtenus en SSE. Dans le cas plan, ils sont respectivement de $\times 2{,}9$ et $\times 2{,}6$ en F32 et F64. Cet écart se retrouve aussi dans le cas cylindrique. La raison pour laquelle le F64 est plus lent n'est pas évidente de prime abord et nécessiterait une meilleure compréhension de l'architecture pour valider ces résultats, d'autant plus qu'une fois parallélisé, ce sont les implémentations F64 qui obtiennent de meilleurs résultats.

Les gains obtenus du fait du passage en F32 sont relativement faibles pour le cas plan, tandis qu'ils sont autour d'un facteur 2 en cylindrique. Cela est dû à la quantité de calcul du cas cylindrique qui est plus importante permettant de mieux profiter de l'accélération de la partie calculatoire SSE. Le facteur 2,7 obtenu en monothread n'est pas non plus simplement explicable et nécessiterait de mieux comprendre l'architecture.

De manière générale, on peut observer un gain allant jusqu'à un facteur 107 lorsque l'on

couple SSE et parallélisation sur 64 threads à l'aide d'OpenMP. Si on observe le gain obtenu entre l'implémentation scalaire F64 monothread et l'implémentation SSE F32 multithreadée, on peut observer un gain avoisinant le facteur 200 (gain non calculé dans les tableaux précédents).

Nous concluons la partie SIMD d'une manière abrupte. Il serait nécessaire d'effectuer plus de benchmark de l'architecture AMD du fait des performances relativement mauvaises obtenues, à la fois en terme de passage à l'échelle en parallélisation, mais aussi dans certains cas SSE où nous n'avons pas d'explications quant à certains comportements. Nous allons maintenant passer à l'analyse générale comprenant les résultats obtenus sur GPP et sur GPU.

3.7 Analyse générale GPP multicoeurs et GPU

Dans cette section, nous allons comparer les résultats obtenus dans les sections précédentes en comparant les résultats GPP et GPU. Nous avons effectué un récapitulatif des transformations apportées à l'algorithme de référence au sein de la figure 3.29. Par manque de temps, seul le groupement de pixels sur GPP n'a pas été réalisé. Cette transformation aurait permis d'améliorer les performances de l'implémentation *Exhaust* sur GPP, mais n'aurait de toutes façons pas pu rattraper l'implémentation *Interp* étant donné que la quantité de calculs sera toujours plus importante en *Exhaust* pour les dimensions de problèmes qui nous intéressent.

Les implémentations basée sur le code SIMD ne sont pas présentes dans la figure citée ci-dessus et ne sont pas comparées dans le tableau principal du fait des différences majeures entre les codes. Les implémentations n'étant pas complètes en termes de fonctionnalités, d'algorithmie et d'optimisations, nous allons comparer les performances obtenues dans un second tableau.

3.7.1 Comparaison GPP / GPU pour les implémentations basées sur le code scalaire CAO2D

Le tableau 3.20 récapitule l'ensemble des performances obtenues pour l'algorithme FTP sur GPP et GPU, pour les différentes implémentations, *Exhaust* et *Interp*, scalaires et gérant la CAO2D, des différentes architectures et des différents jeux de données évalués. Pour rappel, ces performances ont été obtenues en double précision (cf. section 3.4.3.2).

La première remarque que l'on peut faire est simplement que le GPU est systématiquement devant le GPP parallélisé. Les facteurs d'accélération sont variables et se situent entre $\times 1{,}4$ et $\times 4{,}1$. Le facteur $\times 1{,}4$ se retrouve pour DATA 4. Cette faible accélération du GPU est due à la complexité du calcul. En effet, le GPP est beaucoup plus stable dans son comportement lors de la montée en charge en terme de complexité de calcul. Cela parait tout à fait logique dans la mesure où les GPP sont bien mieux parés niveau caches et prédiction de branchements.

De son côté le GPU envoie en cache des valeurs qui devraient être en registre (le nombre de registres physiques étant largement dépassé). Le jeu DATA 4 est d'ailleurs celui où la C2070 est la plus proche des performances de la GTX 580. Cela montre bien que la quantité de calculs nécessaire est très importante et la faible quantité d'unités double précision de la GTX devient limitante. Malgré cela, la GTX reste devant dans l'ensemble des benchmarks, ce qui montre

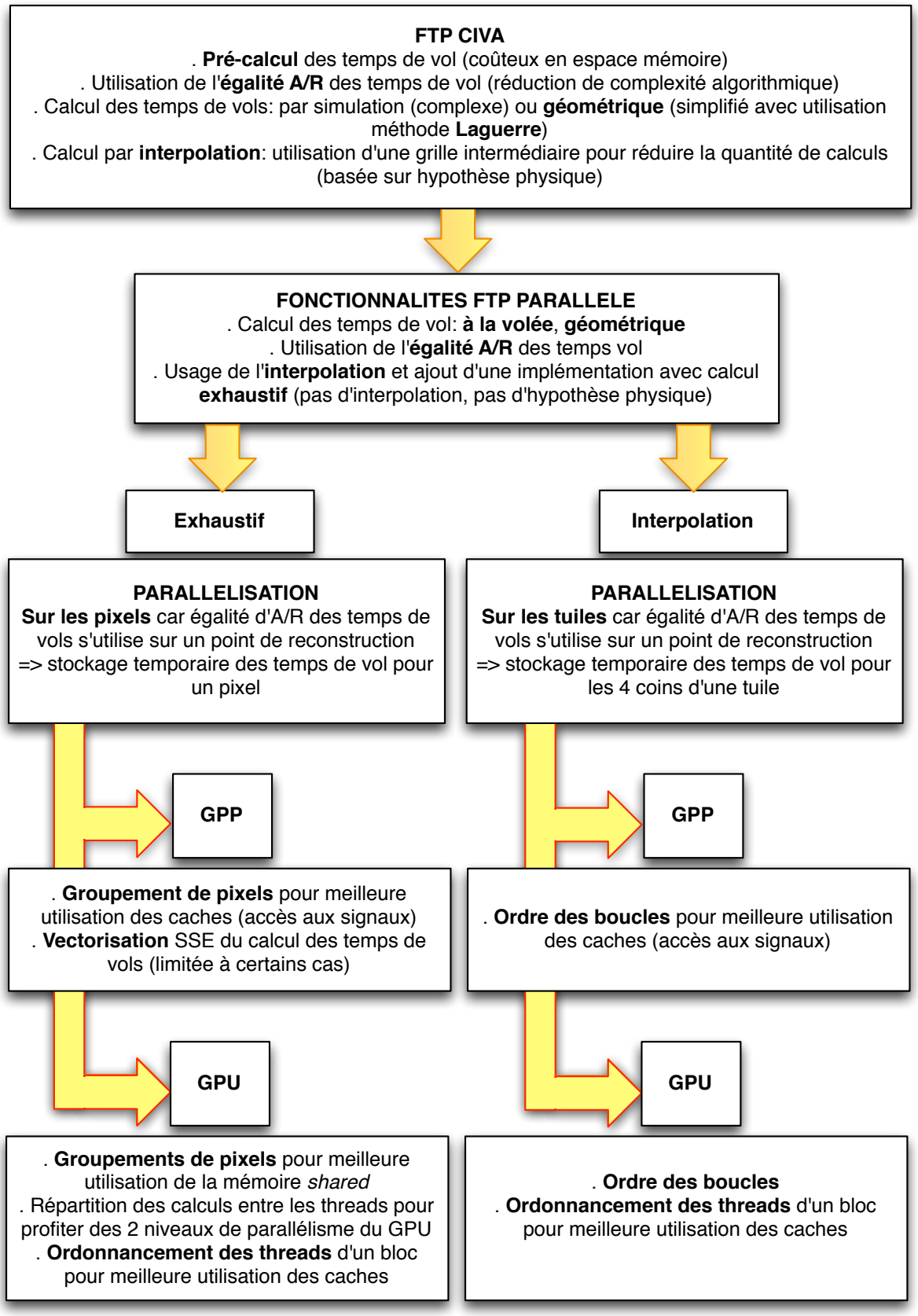


FIGURE 3.29 – Récapitulatif des transformations apportées à l'algorithme de référence.

Tableau 3.20 – Récapitulatif des performances obtenues pour l'algorithme FTP sur GPP et GPU.

Temps (ms)	GPP				GPU				
	OpenMP		OpenCL		CUDA		OpenCL		
Exhaust	**Mono**	**Multi**	**AMD**	**Intel**	**GTX580**	**C2070**	**GTX580**	**C2070**	**HD6970**
DATA 1	10 962	721	1 445	1 667	212	291	238	343	411
DATA 2	27 284	1 699	2 084	2 598	412	555	501	725	667
DATA 3	27 914	1 707	2 182	2 759	458	596	563	806	827
DATA 4	393 340	23 667	21 548	30 758	5 936	6 776	7 121	9 778	9 826
Interp	**Mono**	**Multi**	**AMD**	**Intel**	**GTX580**	**C2070**	**GTX580**	**C2070**	**HD6970**
DATA 1	4 927	446	1 570	2 284	192	227	255	348	381
DATA 2	5 431	473	1 654	2 276	197	235	266	368	384
DATA 3	5 310	460	1 607	2 172	186	212	260	341	391
DATA 4	86 554	5 707	8 649	14 043	2 022	2 158	2 317	3 117	3 642

qu'une carte grand public peut être suffisante, même pour une application utilisant des calculs flottants double précision (une application nécessitant des calculs plus intensifs ferait évidemment changer la balance). Concernant le facteur $\times 4{,}1$, il se retrouve pour les jeux de données intermédiaires en calcul exhaustif. Le GPU dispose d'un nombre de calculs importants, mais suffisamment peu pour ne pas saturer les caches.

En interpolation, les facteurs sont moindres. Cela est dû au coût important des accès en mémoire pour les amplitudes. En effet, le calcul des temps des vols étant peu significatif en temps de calcul, le GPU est contraint de faire un très grand nombre d'accès en mémoire globale, accès qui sont moins efficaces que ceux du GPP qui dispose d'un cache L3. On a donc une perte de performances de l'ordre de 50%, ce qui reste tout à fait raisonnable.

On a donc des GPU qui se placent de manière assez favorable par rapport à la plateforme 2x6 coeurs bien que l'on atteigne pour les jeux de données complexes certaines limites des GPU actuels.

3.7.2 Comparaison GPP / GPU avec les implémentations basées sur le code SIMD

Comparer les valeurs obtenues sur la machine AMD avec celles obtenues sur le code principal, qui pour rappel est scalaire et F64 uniquement, n'est pas chose aisée. Les différences sont multiples et se situent principalement au niveau de la gestion des polynômes de degré inférieur ou égal à 3 et dont la résolution est faite de manière analytique pour le code gérant la CAO2D. Ce dernier utilise aussi un certain nombre de raccourcis à l'aide de branchement pour les cas uniquement réels. S'ajoute à cela les différentes architectures utilisées, qui ne nous permettent pas de comparer directement les résultats. Cependant, malgré les différences, il nous a semblé intéressant de montrer que les écarts sont très réduits comme le présente le tableau 3.21.

On peut observer que les temps sont compris dans un intervalle de facteur 4. Pour des architectures et des implémentations si différentes, ces écarts semblent relativement faibles. Le code reste plus rapide sur GPU bien que la machine AMD à 64 coeurs arrive à un niveau de

Tableau 3.21 – Comparaison de temps de calcul GPP / GPU avec les implémentations basées sur le code SIMD.

Implémentation de base	**CAO2D**	**SIMD**			**CAO2D**
Architecture	**GPP 2x6 coeurs Nehalem**	**GPP 4x16 coeurs Interlagos**			**GPU GTX 580**
Temps (s)	**Scalar F64**	**Scalar F64**	**SSE F64**	**SSE F32**	**F64**
DATA 1	721	881	338	283	212
DATA 2	1 699	2 515	914	499	412

performances quasi équivalent en F32 SIMD.

Un point intéressant est la comparaison avec les temps de calculs avec Interpolation. Dans le tableau 3.20 on peut observer un temps de calcul de 473ms pour DATA2 en Interpolation sur le Nehalem. Ce temps correspond exactement au temps de calcul F32 SIMD sur la machine AMD. Les transformations algorithmiques sont importantes, mais le fait de pouvoir faire passer un calcul à l'échelle sur des machines approchant désormais la centaine de coeurs semble être tout aussi important. Le coût de la transformation de scalaire à SIMD est largement amorti par les gains obtenus.

Nous allons maintenant passer à une discussion sur les développements réalisés et mettre en perspective les résultats obtenus.

3.7.3 Discussion

Nous venons de discuter des performances des implémentations évaluées. Nous avons observé la bonne tenue des GPU malgré leurs limites ainsi que les performances, certes en demi-teinte, mais très encourageantes d'OpenCL.

Plusieurs points qui n'apparaissent pas dans la partie implémentations sont importants à noter. En effet, le développement OpenCL pour FTP n'a pas été aussi simple que pour l'algorithme présenté au chapitre précédent. Les structures de données utilisées par CUDA ne sont pas toutes compatibles avec OpenCL : les vecteurs de données de 3 valeurs sont inutilisables en OpenCL. Les alignements sont faits sur 4 valeurs (4 `float`, 4 `double`, etc...) sans précision du compilateur qui reste la plupart du temps silencieux sur un nombre important d'erreurs. OpenCL dispose toujours de très peu d'outils de débogage ce qui complique extrêmement les développements. Dans le même registre, entre l'implémentation AMD et Intel, nous avons aussi rencontré certains soucis de compatibilité. Nous avons aussi constaté des différences de performances importantes entre deux versions de SDK. Il est donc encore tôt pour considérer OpenCL comme une solution mature. Du côté du GPU, Nvidia se concentrant sur CUDA, OpenCL est laissé en retrait avec une implémentation d'OpenCL 1.1 limitée en fonctionnalités.

Du côté de CUDA, les choses sont plus claires. Les outils continuent de se développer, et l'environnement Nsight pour Visual Studio permet d'obtenir de nombreuses informations. Le passage à LLVM depuis CUDA 4.1 permet d'espérer voir arriver des implémentations CUDA

pour d'autres architectures. Cependant, à l'heure actuelle, seules des API telles que GPU Ocelot ou le compilateur PGI permettent d'exécuter du code CUDA sur d'autres architectures. Nvidia a donc fait un pas en avant vers une utilisation plus générale de son langage.

Enfin, malgré certains comportements parfois peu évidents à comprendre, OpenMP reste le modèle le plus simple à prendre en main. Ici, il nous permet de passer parfaitement à l'échelle dans la plupart des cas. Un calcul de 10 secondes se fait sur un 2x6 coeurs en moins d'une seconde, ce qui représente un gain notable.

L'analyse effectuée nous permet de donner une évaluation des temps de développement en comparant les différents modèles de programmation. Bien sûr, ces estimations dépendent du niveau d'expertise du développeur, du type de code ainsi que d'autres paramètres, mais il est possible de donner une estimation d'ordre général. Si le développement d'un code séquentiel est $1t$, un développement parallélisé via OpenMP pourra prendre $2t$ à $3t$. Quant à un développement GPU sur CUDA, il va pouvoir prendre environ $5t$ pour obtenir un code fonctionnel et jusqu'à $15t$ pour optimiser ce code. Enfin, un développement OpenCL peut même aller jusqu'à $20t$, dans la mesure où l'objectif sera d'obtenir un code qui soit utilisable sur différentes plateformes (ce qui reste difficile à faire sans dupliquer certaines parties du code). Ces coûts importants sont à chiffrer, et à comparer au besoin. Aujourd'hui, les performances obtenues en OpenMP sont déjà très satisfaisantes et devraient continuer à passer à l'échelle sur les architectures multicoeurs à court terme.

3.8 Intégration de l'algorithme FTP dans la plateforme CIVA

L'algorithme FTP a été intégré sous forme de prototype dans la plateforme CIVA. Cette intégration a été réalisée de la même manière que pour l'algorithme IVC étudié dans le chapitre 2. La gestion de la partie hôte réutilise les classes développées pour IVC, et les aspects transferts de données sont gérés de la même manière, c'est à dire que les données sont considérées comme résidentes en mémoire de l'accélérateur afin d'éviter les transferts de gros volumes de données.

3.8.1 Présentation du prototype

L'étude réalisée sur FTP a été faite pour des cas de reconstructions 2D. En effet, ce sont les plus utilisées car les plus rapides à reconstruire et les moins consommatrices en mémoire dans l'algorithme historique avec précalcul des temps de vol. Néanmoins, il est tout à fait possible et pertinent pour l'utilisateur d'utiliser des zones 3D. Une extension du code OpenCL *Interp* a donc été réalisée afin de prendre en compte ces zones.

La figure 3.30 présente un schéma du travail effectué. Le passage en 3D permet ensuite d'utiliser les outils de visualisation de CIVA pour afficher, par exemple, des iso-surfaces à partir d'une grille 3D.

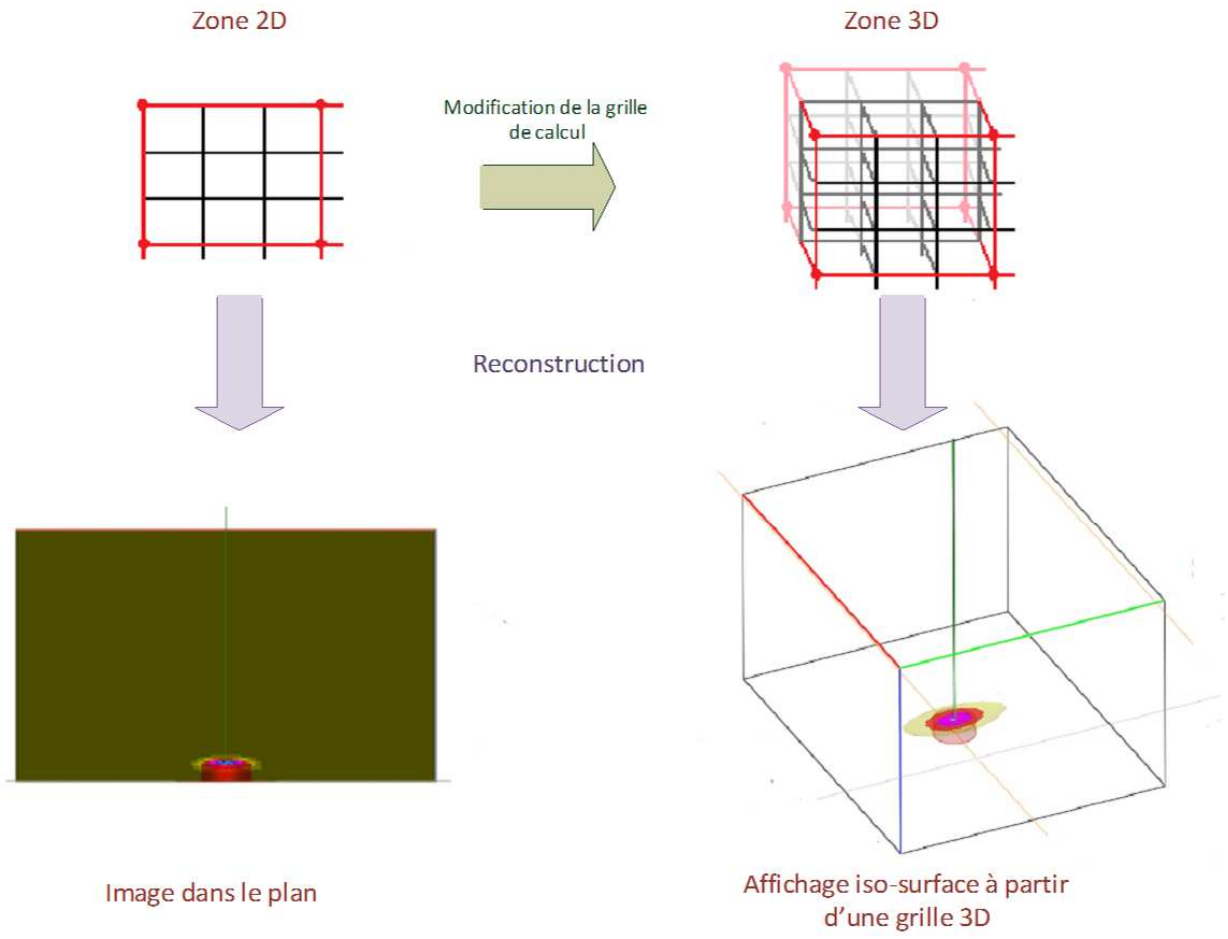


FIGURE 3.30 – Extension des reconstructions FTP aux cas 3D.

3.8.2 Benchmark

Nous avons réalisé un ensemble de benchmarks pour valider les performances de l'intégration. Le tableau 3.22 présente les performances de reconstructions FTP pour le jeu de données DATA 1 et des images de dimension 400×500 en 2D et un volume de 100^3 en 3D. La machine utilisée est la même que pour les benchmarks réalisés sur IVC (cf. section 2.8.2).

Les écarts de performances entre GPP et GPU sont équivalents à ceux hors CIVA. Le code optimisé pour GPU n'est pas du tout efficace sur GPP et les performances sont donc très mauvaises. En revanche, on peut noter que par rapport à CIVA 10, les performances sont largement améliorées. Le GPU permet d'obtenir une accélération d'un facteur $\times 70$ en 2D et $\times 54$ en 3D. Les raisons de ces facteurs importants sont multiples mais sont principalement le fait que l'accès aux données n'est pas multithreadé dans l'implémentation CIVA 10, ne permettant pas un passage à l'échelle correct.

Tableau 3.22 – Performances de l'intégration FTP dans CIVA en 2D et 3D.

temps (ms)	FTP 2D	FTP 3D
OpenCL GPU (C2070)	570	12 270
OpenCL GPP	12 016	172 338
CIVA 10 (C++ multithreadé)	39 359	673 055
Gain GPU/CIVA 10	$\times 69$	$\times 54$
Gain OpenCL GPU/GPP	$\times 21$	$\times 14$

Si le code OpenCL n'est pas vraiment exploitable sur GPP, ce prototype aura permis de montrer combien il est nécessaire d'améliorer l'accès aux données pour obtenir de bonnes performances de reconstruction pour cet algorithme. En effet, les temps de reconstruction obtenus sur GPU sont très intéressants avec un temps reconstruction 2D sous la seconde et de l'ordre de la dizaine de secondes en 3D.

L'intégration de l'algorithme FTP permet, au travers de la modification d'ergonomie pour l'utilisateur, d'obtenir des reconstructions dans des temps plus exploitables. En 2D déjà, de passer de plusieurs minutes de reconstruction pour un cas simple, à un temps de l'ordre de la seconde sur GPU, est tout à fait probant.

C'est bien sûr en 3D que l'intérêt des accélérations est d'autant plus important. Le fait de passer de plusieurs minutes de calcul à quelques secondes devrait permettre aux utilisateurs d'utiliser plus régulièrement ces reconstructions, alors que jusqu'ici, elles n'étaient utilisées que pour des cas très particuliers, du fait des temps de calculs importants.

Nous sommes bien sûr conscients du fait que, selon la configuration, nous pourrons très bien retrouver des temps de calcul dépassant la minute en 3D, mais nous avons fait le choix d'utiliser une résolution de taille moyenne par rapport aux besoins habituels des utilisateurs. Cela permet d'avoir une idée assez juste des ordres de grandeurs des temps de reconstruction et du gain apporté par la parallélisation pour l'utilisateur.

3.9 Conclusion

Dans ce chapitre, nous avons présenté l'étude d'un second algorithme de reconstruction ultrasons basé sur la notion de temps de vol. A contrario de l'algorithme vu au chapitre précédent, ces temps de vol n'étaient pas une entrée, mais devaient être calculés à la volée.

Nous avons effectué l'analyse du code existant afin d'en extraire les fonctionnalités principales et analysé la méthode de calcul. Nous avons listé trois méthodes de calcul et évalué les deux premières ; la troisième étant une extension de la méthode de calcul par interpolation des temps de vol. Nous avons proposé différentes approches de parallélisation pour les deux méthodes de calcul et nous les avons implémentées.

Notre algorithme de base étant en double précision et les seules sources que nous avons trouvées nous indiquant que les algorithmes de recherche de zéro robustes en simple précision

ne sont pas courants, nous avons réalisé nos implémentations en double précision. Cependant, nous avons souhaité étudier la possibilité d'exploiter la simple précision. Pour cela, nous avons utilisé deux approches nous montrant que sous certaines conditions, la méthode de recherche de zéro est exploitable pour le type de conditionnement de polynômes que nous avons et ce, pour les ordres les plus importants nécessaires dans FTP.

Nous avons ensuite évalué les performances des implémentations double précision sur plusieurs architectures GPU et GPP multicoeurs au travers d'un grand nombre de benchmarks. Ceux-ci nous ont permis, en se basant sur des jeux de données typiques de ce que l'utilisateur final de FTP peut rencontrer, d'effectuer l'analyse des différents codes et différentes architectures. Nous avons constaté que la méthode de calcul par interpolation, utilisé par l'implémentation de référence n'est pas toujours la plus performante. En effet, la parallélisation sur les GPP multicoeurs implique une plus forte pression sur les différents niveaux de cache et sur les accès mémoire en général ce qui limite, dans les cas avec peu de calculs, les performances des GPP comme avec les reconstructions de surfaces planes simples. Cette observation se retrouve aussi sur GPU. En revanche, dès lors que le nombre de surfaces augmente, ou simplement que le type de surface implique un calcul plus complexe comme pour les surfaces cylindriques, nous constatons que les implémentations redeviennent rapidement limitées par le calcul.

Les benchmarks simple précision ayant été réalisés à postériori, nous n'avons pu les intégrer dans la globalité. Cependant, nous montrons que la vectorisation SSE permet d'accélérer de manière très efficace l'algorithme de Laguerre et par conséquent, d'obtenir de très bonnes performances pour les cas les plus couteux en calcul.

L'évaluation de l'API OpenCL nous a à nouveau appris que sur GPU, les performances sont toujours en deçà de CUDA sur les GPU Nvidia. Nous avons pu constater aussi la bonne tenue de la AMD HD6970 qui, sans être de la toute dernière génération, réussit à atteindre des performances proches de la Tesla C2070. Quant aux implémentations pour GPP multicoeurs, le niveau de performance est encore limité avec dans le meilleur des cas évalués, jusqu'à 40% de performance en moins qu'OpenMP parallélisé sur 24 threads.

Concernant les différences entre implémentations par calcul exhaustif et par interpolation des temps de vol, nous avons pu constater que l'interpolation est de manière générale très efficace. Malgré cela, dans certains cas sur GPP multicoeurs avec peu de calculs, on constate qu'une fois parallélisé, le passage à l'échelle est moins bon et de meilleures performances sont obtenues en calcul exhaustif qu'en interpolation. Avec l'augmentation du nombre de coeurs et l'arrivée d'architectures telles que l'Intel Xeon Phi ou simplement les quad-socket aussi chez Intel, certaines techniques plus coûteuses en calculs mais plus régulières et moins demandeuses en accès mémoires vont redevenir plus performantes.

L'intégration de l'algorithme FTP dans la plateforme CIVA aura permis de confirmer la nécessité de travailler sur l'accès aux données pour obtenir de bonnes performances en parallélisation. Le prototype permet d'obtenir des reconstructions sur GPU avec des temps de calcul jusqu'à deux ordres de grandeur inférieurs à ceux de la version commerciale de CIVA 10. De cette manière, les reconstructions 3D, jusqu'ici très coûteuses, deviennent accessibles aux utilisateurs.

Conclusion générale

Les travaux exposés dans ce manuscrit se placent à la jonction entre le domaine scientifique du Contrôle Non Destructif par Ultrasons (CND US) et l'Adéquation Algorithme Architecture. De manière générale, l'objectif du CND est de rechercher d'éventuels défauts dans des pièces en production ou en maintenance. Pour se faire, des algorithmes de reconstruction sont utilisés. Ceux-ci nécessitent généralement de pouvoir obtenir le résultat rapidement. On parle alors de temps de calcul interactif.

Le matériel d'acquisition évoluant, les volumes de données à traiter sont toujours plus importants. Les architectures de calcul évoluent elles-aussi. Néanmoins, le tournant du parallélisme, pris par les GPP (*General Purpose Processor*) grand-public autour de 2006, n'a pas permis de continuer à améliorer significativement les performances de reconstruction. En effet, la plupart des algorithmes existants ont été développés pour être exécutés de manière séquentielle. L'arrivée du parallélisme nécessite donc de faire évoluer ces algorithmes en fonction des architectures existantes. Dans le cadre de l'accélération de calculs de reconstruction ultrasons pour la plateforme logicielle de CND CIVA, nous nous sommes intéressés à la parallélisation de deux algorithmes de reconstruction très utilisés. Pour autant, ils sont limités dans leurs possibilités d'utilisation par des temps de calcul non interactif. Les architectures ciblées sont les deux architectures les plus adaptées à l'intégration dans un contexte industriel et visant des machines de type *desktop* : les GPP et les GPU (*Graphics Processing Unit*). Ces architectures sont aujourd'hui programmables à l'aide de différents outils et langages. OpenMP a été utilisé sur GPP tandis que CUDA a été évalué pour les GPU Nvidia. Nous avons aussi évalué, au travers des implémentations AMD et Intel, la capacité d'OpenCL à s'adapter sur GPP pour des codes développés initialement pour GPU.

Le premier algorithme étudié se base sur la notion de trajet, approximation du chemin de propagation de l'onde ultrasonore dans un milieu. L'algorithme consiste en une projection par un calcul de maximum d'amplitude le long de ces trajets. Un traitement d'image par dilatation était initialement effectué à la volée et a été remplacé par une dilatation en post-traitement, permettant d'améliorer significativement les performances de l'algorithme. Concernant la parallélisation, le schéma adopté sur GPP permet de s'affranchir de la concurrence entre les threads, en allouant un espace mémoire propre aux threads. Sur GPU, la grande quantité de threads ne permet pas cette optimisation et l'utilisation d'instructions atomiques est nécessaire.

Le passage à l'échelle est excellent et l'*Hypertheading* des GPP Intel permet, sur une machine équipée de deux processeurs 6 coeurs, d'obtenir un facteur d'accélération surlinéaire de l'ordre de $\times 16$ par rapport à une exécution séquentielle. De plus la parallélisation GPP permet d'avoir des performances stables entre les jeux de données et les différentes vues de l'algorithme. A contrario, les performances du GPU sont dépendantes de ces paramètres. Les performances obtenues sur un GPU Nvidia haut de gamme de génération Fermi permettent d'obtenir des

performances du même ordre de grandeur que celles obtenues sur GPU. Le manque d'efficacité des GPU est principalement dû aux écritures aléatoires qui n'ont pas pu être améliorées. L'utilisation d'OpenCL pour cet algorithme a montré que les performances sur GPU étaient très proches de celles obtenues avec un outil natif. Quant aux performances sur GPP, on a pu constater en moyenne des performances de l'ordre de 30% inférieures, laissant penser que malgré la jeunesse des implémentations d'Intel et AMD, OpenCL puisse être une alternative intéressante pour une intégration dans CIVA.

Le second algorithme étudié se base sur la notion de temps de vol. Contrairement au premier algorithme, les temps de vols nécessaires à la reconstruction sont un facteur limitant pour la plateforme CIVA nécessitant un stockage mémoire relativement coûteux. Nous nous sommes donc intéressés à transformer cet algorithme pour prendre en compte un calcul des temps de vol à la volée. Ce calcul étant basé sur une méthode de résolution d'équations polynomiales, nous avons implémenté la méthode de Laguerre et utilisé celle-ci pour nos implémentations GPP et GPU.

Deux méthodes de calcul ont été évaluées, l'une effectuant un calcul systématique des temps de vol, et l'autre utilisant une réduction des calculs grâce à une interpolation se basant sur une hypothèse physique valide dans une majorité de cas. Une optimisation connue de l'algorithme étudiée implique une certaine localité dans le calcul des temps de vol. Pour l'exploiter, il est donc nécessaire d'utiliser un schéma de parallélisation précis que nous avons utilisé à la fois sur GPP et GPU. Pour obtenir la meilleure utilisation des caches et un bon ordonnancement dans les accès à la mémoire, on s'est intéressé à l'ordre des nids de boucles sur GPP et à l'utilisation de mémoire partagée et à l'ordonnancement des threads du GPU.

L'algorithme de référence ayant été historiquement développé en double précision, nous avons réalisé la majeure partie de notre étude dans ce cas. Cependant, nous avons étudié la possibilité d'exécuter les calculs en simple précision de manière à pouvoir mieux profiter des unités vectorielles des GPP et de la puissance de calcul des GPU. A l'aide de deux approches, nous avons validé la stabilité du code sur une majorité des cas d'utilisation et réalisé une implémentation SSE que nous avons évalué sur un sous-ensemble des jeux de données. Cette implémentation montre qu'il est possible d'accélérer significativement l'algorithme et ce, malgré sa structure itérative.

Les performances obtenues à l'aide des langages natifs sont excellentes, avec un bon passage à l'échelle des GPP et des GPU. Ces derniers sont de 2 à 5 fois plus rapides que les GPP. L'utilisation d'OpenCL sur GPU s'est avérée tout à fait efficace. En revanche, l'utilisation sur GPP des codes OpenCL GPU s'est avérée très décevante, ne réalisant que des accélérations très faibles par rapport au code GPP monothreadé. Ces performances s'expliquent par la nécessité d'utiliser un ordonnancement des accès à la mémoire spécifique à chaque architecture, dans le cas d'un algorithme limité par les accès mémoires comme celui-ci.

Les deux algorithmes étudiés ont tous deux été portés sous forme de prototype dans CIVA. Un ensemble de problématiques ont été mises en évidence, telle que celle de l'évolution de codes optimisés pour le premier algorithme. L'utilisation d'instructions atomiques s'est avéré problématique lorsque nous avons voulu étendre les fonctionnalités de l'algorithme. Ce n'est que par un contournement que nous avons pu finaliser l'intégration. Les performances par rapport à la version commerciale de CIVA sont bien sûr très flatteuses, permettant des accélérations allant jusqu'à un facteur $\times 150$, mais celles-ci sont à relativiser dans la mesure où

l'exécution était monothreadée. Cet algorithme a tout de même été très nettement amélioré, à la fois sur le plan algorithmique et en terme de parallélisation permettant de réduire significativement les temps de reconstruction et permettant d'obtenir, pour des configurations de taille importante, des temps de reconstruction interactif. Quant au second algorithme, une extension des fonctionnalités, avec la gestion de reconstructions 3D, montre elle-aussi des gains importants avec des accélération de l'ordre d'un facteur $\times 50$.

Les gains importants observés sont à relativiser dans la mesure ou l'une des sources principales est l'accès aux données. En effet, les prototypes intégrés chargent l'ensemble des données en DRAM avant de les exploiter. Ce système n'étant pas adopté dans la plateforme CIVA, une relecture à la volée est nécessaire et peut s'avérer très coûteuse. C'est donc un des premiers points sur lesquels il est nécessaire de travailler pour pouvoir bénéficier d'accélérations grâce à la parallélisation.

Quant au choix architectural à effectuer, il semble que les GPP soient actuellement des solutions tout à fait viables, à moins d'avoir un objectif chiffré et d'avoir besoin des petits gains que l'on peut obtenir sur GPU. En effet, lorsque l'on place en balance un facteur $\times 5$ d'accélération par rapport à un GPP face au coût et au temps de développement nécessaire pour l'obtenir, il semble difficile de s'orienter dans cette direction. En revanche, si l'installation de GPU peut être réalisée sur des machines de types dual ou quad-core, le facteur $\times 5$ peut se transformer en facteur $\times 10$ à $\times 20$ et le GPU retrouve de son intérêt. Il est donc nécessaire de faire un état des lieux et d'évaluer le besoin avant d'entreprendre des portages sur ces architectures.

Nous pouvons retenir de nos développements qu'il est intéressant de procéder dans l'ordre par une étude algorithmique, puis un portage avec un outil tel qu'OpenMP qui permet rapidement d'évaluer les capacités de parallélisation sur GPP. Si les performances obtenues ne sont pas suffisantes, ou que la cible est du GPU, il est intéressant dans un second temps d'effectuer un portage CUDA pour profiter de l'environnement de développement et des outils mis à disposition. Enfin, un portage OpenCL peut être fait très rapidement pour les GPU puisque les deux API sont très proches. L'utilisation d'OpenCL sur GPP est peu convaincante pour le moment même s'il semble toujours aussi intéressant de pouvoir exécuter un même code sur deux architectures aussi différentes que GPP et GPU. L'utilisation d'un langage tel qu'OpenACC pourra être intéressant dans cette mesure pour permettre – sans duplication importante de code – de spécifier certains niveaux de parallélisation pour une architecture particulière. Le standard venant d'apparaitre, il n'est pas encore pertinent d'intégrer des développements complets avec cet outil. En revanche, il est à suivre de près.

Les travaux réalisés dans cette thèse vont pouvoir être réutilisés. En effet, les algorithmes étudiés peuvent être étendus en termes de fonctionnalités, sans pour autant avoir à remettre en question les schémas de parallélisation. La brique de base qu'est le calcul des temps de vol va pouvoir être réutilisée dans le cadre d'une thèse ayant débuté en 2012 au sein du CEA-LIST/DISC en vue de l'accélération, non plus de post-traitements mais du calcul de simulation de champ et d'échos. L'objectif de cette thèse n'est plus d'obtenir des temps interactifs mais d'obtenir une simulation en temps-réel. Quant aux algorithmes en eux-mêmes, ils pourront être réutilisés sur des nouvelles architectures telles que le Xeon Phi d'Intel grâce à OpenMP et à l'implémentation SIMD de FTP. De plus, les schémas de parallélisation pourront être réutilisés en vue de parallélisation à l'aide de nouveaux outils tels qu'OpenACC.

RÉFÉRENCES BIBLIOGRAPHIQUES

[Adams, 1967] Adams, D. A. (1967). A stopping criterion for polynomial root finding. *Commun. ACM*, 10(10) :655–658.

[Chesneaux, 1995] Chesneaux, J.-M. (1995). *L'arithmétique Stochastique et le Logiciel CADNA, Habilitation à diriger les recherches*. PhD thesis, Université Pierre et Marie Curie, Paris.

[Courbin et al., 2009] Courbin, P., Saidani, T., and Lacassagne, L. (2009). Parallélisation d'opérateurs de TI : multi-coeurs, Cell ou GPU ? *XXIIe colloque GRETSI (traitement du signal et des images), Dijon (FRA), 8-11 septembre 2009*.

[Curie and Curie, 1880] Curie, P. and Curie, J. (1880). Développement par pression de l'électricité polaire dans les hémiédries à faces inclinées. *Comptes Rendus de l'Académie des Sciences*, 91 :294–295.

[Domanski et al., 2009] Domanski, L., Vallotton, P., and Wang, D. (2009). Parallel van Herk/Gil-Werman image morphology on GPUs using CUDA. GTC 2009.

[Frey, 2012] Frey, J. (2012). Mills Performance Testing : Threading Efficiency. Technical report, IT-NSS, University of Delaware.

[Gac et al., 2008] Gac, N., Mancini, S., Desvignes, M., and Houzet, D. (2008). High speed 3D tomography on CPU, GPU, and FPGA. *EURASIP J. Embedded Syst.*, 2008 :5 :1–5 :12.

[Holmes et al., 2004] Holmes, C., Drinkwater, B., and Wilcox, P. (2004). The post-processing of ultrasonic array data using the total focusing method. *Insight - Non-Destructive Testing and Condition Monitoring*, 46(11) :677–680.

[Holmes et al., 2005] Holmes, C., Drinkwater, B. W., and Wilcox, P. D. (2005). Post-processing of the full matrix of ultrasonic transmit-receive array data for non-destructive evaluation. *NDT & E International*, 38(8) :701 – 711.

[Holmes et al., 2008] Holmes, C., Drinkwater, B. W., and Wilcox, P. D. (2008). Advanced post-processing for scanned ultrasonic arrays : Application to defect detection and classification in non-destructive evaluation. *Ultrasonics*, 48(6-7) :636 – 642. Selected Papers from ICU 2007.

[Hutter, 2012] Hutter, M. (2012). Java bindings for OpenCL (website). `http://www.jocl.org/`.

[Jenkins, 1975] Jenkins, M. A. (1975). Algorithm 493 : Zeros of a real polynomial [c2]. *ACM Trans. Math. Softw.*, 1(2) :178–189.

[Jobst and Connolly, 2010] Jobst, M. and Connolly, G. D. (2010). Demonstration of the Application of the Total Focusing Method to the Inspection of Steel Welds. In *10th European Conference on Non-Destructive Testing*.

[Kajiya, 1982] Kajiya, J. T. (1982). Ray tracing parametric patches. *SIGGRAPH Comput. Graph.*, 16(3) :245–254.

[Kirk and Hwu, 2010] Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Mmassively Parallel Processors : A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Laguerre, 1879] Laguerre, E. (1879). Sur le développement d'une fonction suivant les puissances croissantes d'un polynôme. *Journal für die reine und angewandte Mathematik*, 88 :35–48.

[Lévesque et al., 2002] Lévesque, D., Blouin, A., Néron, C., and Monchalin, J.-P. (2002). Performance of laser-ultrasonic F-SAFT imaging. *Ultrasonics*, 40(10) :1057 – 1063.

[Mekwi, 2001] Mekwi, W. (2001). *Iterative Methods for Roots of Polynomials*. Oxford University.

[Nvidia, 2009] Nvidia (2009). Whitepaper, Nvidia's Next Generation CUDA Compute Architecture : Fermi. Technical report, Nvidia Corporation.

[Paragios et al., 2005] Paragios, N., Chen, Y., and Faugeras, O. (2005). *Handbook of Mathematical Models in Computer Vision*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Podlozhnyuk, 2007] Podlozhnyuk, V. (2007). Image convolution with cuda. technical report, Nvidia.

[Porre et al., 2005] Porre, J., Mahaut, S., Chatillon, S., and Calmon, P. (2005). Simulation of Phased Array Techniques and Model Based Data Reconstruction. *AIP Conference Proceedings*, 760(1) :906–913.

[Press et al., 2002] Press, W. H., Vetterling, W. T., Teukolsky, S. A., and Flannery, B. P. (2002). *Numerical Recipes in C++ : the art of scientific computing*. Cambridge University Press, New York, NY, USA, 2nd edition.

[Pédron et al., 2012] Pédron, A., Lacassagne, L., Barbillon, V., Bimbard, F., Rougeron, G., and Berre, S. L. (2012). Performance Analysis of an Ultrasound Reconstruction Algorithm for Non Destructive Testing. *Advances in Parallel Computing*, 22 :261–268.

[Pédron et al., 2011] Pédron, A., Lacassagne, L., Bimbard, F., and Berre, S. L. (2011). Parallelization of an ultrasound reconstruction algorithm for non destructive testing on multicore CPU and GPU. In *Proceedings of the 2011 Conference on Design and Architectures for Signal and Image Processing*, pages 347–354.

[Pédron et al., 2010] Pédron, A., Laguzet, F., Saidani, T., Courbin, P., Lacassagne, L., and Gouiffès, M. (2010). Parallélisation d'opérateurs de ti : multi-cœurs, Cell ou GPU ? *TS. Traitement du signal*, 27(2) :161–187.

[Roerdink, 2003] Roerdink, J. B. T. M. (2003). Multiresolution maximum intensity volume rendering by morphological adjunction pyramids. *IEEE Trans Image Process*, 12(6) :653–60.

[Romero-Laorden et al., 2011a] Romero-Laorden, D., Martinez-Graullera, O., Martín, C. J., Pérez, M., and Ullate, L. G. (2011a). Field modelling acceleration on ultrasonic systems using graphic hardware. *Computer Physics Communications*, 182(3) :590–599.

[Romero-Laorden et al., 2011b] Romero-Laorden, D., O.Martínez-Graullera, C.J.Martín-Arguedas, M.Pérez, and L.G.Ullate (2011b). Paralelización de los procesos de conformación de haz para la implementación del total focusing method. In *CAEND Comunicaciones congresos, 12º Congreso Español de Ensayos No Destructivos*.

[Royer. and E.Dieulesaint, 1996] Royer., D. and E.Dieulesaint (1996). *Ondes élastiques dans les solides, Tome 1*. Masson, Paris, France, 1st edition.

[Saidani et al., 2011] Saidani, T., Lacassagne, L., Falcou, J., Tadonki, C., and Bouaziz, S. (2011). Parallelization Schemes for Memory Optimization on the Cell Processor : A Case Study on the Harris Corner Detector. *T. HiPEAC*, 3 :177–200.

[Scott et al., 2007] Scott, N. S., Jézéquel, F., Denis, C., and Chesneaux, J.-M. (2007). Numerical 'health check' for scientific codes : the CADNA approach. *Computer Physics Communications*, 176(8) :507–521. PEQUAN LIP6.

[Seydel, 1982] Seydel, J. (1982). Ultrasonic synthetic-aperture focusing techniques in NDT. *Research Techniques in Nondestructive Testing*, 6 :1–47.

[Sutcliffe et al., 2012] Sutcliffe, M., Weston, M., Dutton, B., Charlton, P., and Donne, K. (2012). Real-time full matrix capture for ultrasonic non-destructive testing with acceleration of post-processing through graphic hardware. *NDT & E International*, 51(0) :16 – 23.

[Torres et al., 2012] Torres, Y., Gonzalez-Escribano, A., and Llanos, D. R. (2012). Using Fermi architecture knowledge to seeped up CUDA and OpenCL programs. In *International Workshop on Heterogeneous Architectures and Computing*.

[van Herk, 1992] van Herk, M. (1992). A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recogn. Lett.*, 13 :517–512.

[Volkov and Demmel, 2008] Volkov, V. and Demmel, J. W. (2008). Benchmarking GPUs to Tune Dense Linear Algebra. In *SC '08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA. IEEE Press.

[Wang et al., 2011] Wang, H., Desbat, L., and Legoupil, S. (2011). "Image representation by blob and its application in CT reconstruction from few projections". *ArXiv e-prints*.

[Weissten, 2012] Weissten, E. W. (2012). Laguerre's method. From MathWorld - A Wolfram Web Ressource, `http://mathworld.wolfram.com/LaguerresMethod.html`.

[Wilkinson, 1959] Wilkinson, J. H. (1959). The evaluation of the zeros of ill-conditioned polynomials. Part I. *Numerische Mathematik*, 1(1) :150–166.

[Williams et al., 2005] Williams, A., Barrus, S., Morley, R. K., and Shirley, P. (2005). An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA. ACM.

[Wong et al., 2010] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. (2010). Demystifying GPU Microarchitecture through Microbenchmarking. *ISPASS*, pages 235–246.

[Y. Tasinkevych, 2010] Y. Tasinkevych, A. Nowicki, I. T. (2010). Element directivity influence in the synthetic focusing algorithm for ultrasound imaging. *Proceedings of the LVII Open Seminar on Acoustics*, pages 197–200.

Publications

Revue nationale

TS 2010, "Parallélisation d'opérateurs de TI : multi-coeurs, Cell ou GPU" A. Pédron, F. Laguzet, T. Saidani, P. Courbin, L. Lacassagne, M. Gouiffès, Traitement du Signal, 2010.

Conférences Internationales

DASIP 2012, "Analysis of multicore CPU and GPU toward parallelisation of Total Focusing Method ultrasound reconstruction", J. Lambert, A. Pédron, G. Gens, F. Bimbard, L. Lacassagne, E. Iakovleva, S. Le Berre, IEEE International Conference on Design and Architectures for Signal and Image Processing, 2012.

DASIP 2011, "Parallelization of an ultrasound reconstruction algorithm for non destructive testing on multicore CPU and GPU", A. Pédron, L. Lacassagne, F. Bimbard, S. Le Berre, IEEE International Conference on Design and Architectures for Signal and Image Processing, 2011, Best Poster Award.

PARCO 2011, "Performance analysis of an ultrasound reconstruction algorithm for non destructive testing", A. Pédron, L. Lacassagne, V. Barbillon, F. Bimbard, G. Rougeron, S. Le Berre, IEEE International Conference on Parallel Computing, 2011.

Conférence nationale

GRETSI 2009, "Parallélisation d'opérateurs de TI : multi-coeurs, Cell ou GPU", P. Courbin, A. Pédron, T. Saidani, L. Lacassagne, Groupe d'Etudes du Traitement du Signal et des Images, 2009.

Ecole d'été

EEFTIG 2011, "Contrôle non destructif : implantation d'algorithmes sur GPU et multicoeurs", G. Rougeron, A. Pédron, Ecole d'Eté Francophone de Traitement d'Image sur GPU, 2011.

EEFTIG 2011, "Techniques d'optimisations sur Multi-coeurs et GPU pour le Traitement d'Images Haute Performance", L. Lacassagne, A. Pédron, Ecole d'Eté Francophone de Traitement d'Image sur GPU, 2011.

EEFTIG 2010, "Parallélisation d'opérateurs de TI : multi-coeurs, Cell ou GPU", L. Lacassagne, A. Pédron, F. Laguzet, M. Gouiffès, Ecole d'Eté Francophone de Traitement d'Image sur GPU, 2010.

EETIG 2010, "Implantation d'algorithmes d'*inpainting* sur GPU", A. Pédron, Y. Moudden, P. Kestener, Ecole d'Eté Francophone de Traitement d'Image sur GPU, 2010.

Résumé

La problématique de cette thèse se place à l'interface entre le domaine scientifique du contrôle non destructif par ultrasons (CND US) et l'adéquation algorithme-architecture. Le CND US comprend un ensemble de techniques utilisées pour examiner un matériau, qu'il soit en production ou maintenance. Afin de détecter d'éventuels défauts, de les positionner et les dimensionner, des méthodes d'imagerie et de reconstruction ont été développées au CEA-LIST, dans la plateforme logicielle CIVA.

L'évolution du matériel d'acquisition entraine une augmentation des volumes de données et par conséquent nécessite toujours plus de puissance de calcul pour parvenir à des reconstructions en temps interactif. L'évolution multicoeurs des processeurs généralistes (GPP), ainsi que l'arrivée de nouvelles architectures comme les GPU rendent maintenant possible l'accélération de ces algorithmes.

Le but de cette thèse est d'évaluer les possibilités d'accélération de deux algorithmes de reconstruction sur ces architectures. Ces deux algorithmes diffèrent dans leurs possibilités de parallélisation. Pour un premier, la parallélisation sur GPP est relativement immédiate, contrairement à celle sur GPU qui nécessite une utilisation intensive des instructions atomiques. Quant au second, le parallélisme est plus simple à exprimer, mais l'ordonnancement des nids de boucles sur GPP, ainsi que l'ordonnancement des threads et une bonne utilisation de la mémoire partagée des GPU sont nécessaires pour obtenir un fonctionnement efficace. Pour ce faire, OpenMP, CUDA et OpenCL ont été utilisés et comparés. L'intégration de ces prototypes dans la plateforme CIVA a mis en évidence un ensemble de problématiques liées à la maintenance et à la pérennisation de codes sur le long terme.

Abstract

This thesis work is placed between the scientific domain of ultrasound non-destructive testing and algorithm-architecture adequation. Ultrasound non-destructive testing includes a group of analysis techniques used in science and industry to evaluate the properties of a material, component, or system without causing damage. In order to characterise possible defects, determining their position, size and shape, imaging and reconstruction tools have been developed at CEA-LIST, within the CIVA software platform.

Evolution of acquisition sensors implies a continuous growth of datasets and consequently more and more computing power is needed to maintain interactive reconstructions. General purpose processors (GPP) evolving towards parallelism and emerging architectures such as GPU allow large acceleration possibilities than can be applied to these algorithms.

The main goal of the thesis is to evaluate the acceleration than can be obtained for two reconstruction algorithms on these architectures. These two algorithms differ in their parallelization scheme. The first one can be properly parallelized on GPP whereas on GPU, an intensive use of atomic instructions is required. Within the second algorithm, parallelism is easier to express, but loop ordering on GPP, as well as thread scheduling and a good use of shared memory on GPU are necessary in order to obtain efficient results. Different API or libraries, such as OpenMP, CUDA and OpenCL are evaluated through chosen benchmarks. An integration of both algorithms in the CIVA software platform is proposed and different issues related to code maintenance and durability are discussed.